REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-98-

0374

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT
	January 1996	Final

4. TITLE AND SUBTITLE
Methodologies for Built-In Self-Test Insertion in VLSI Circuits Across the Design Hierarchy

5. FUNDING NUMBERS

6. AUTHORS
Joan Elizabeth Carletta

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Case Western Reserve University

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NI
110 Duncan Avenue, Room B-115
Bolling Air Force Base, DC 20332-8080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
See attached.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

19980430 027

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

METHODOLOGIES FOR BUILT-IN SELF-TEST INSERTION IN VLSI CIRCUITS ACROSS THE DESIGN HIERARCHY

Abstract

by

JOAN ELIZABETH CARLETTA

Methodologies for built-in self-test (BIST) insertion in VLSI circuits are presented for three different levels of design abstraction. The methodologies are designed to be used during the design flow of application specific integrated circuits (ASICs), which starts at the algorithmic level in the behavioral domain and moves to the register transfer level in the structural domain, and finally to the gate level in the structural domain. At each level, the methodology is based on the use of testability metrics to identify and remove points of low testability in a circuit. By quantifying the properties that make a BIST scheme successful, the testability metrics provide a way to measure test quality implicitly, without resorting to fault simulation, which is both expensive and not available at the higher levels of design abstraction. The testability metrics are computed using a Markov chain model.

Fault coverage curves show that when the BIST insertion methodologies are applied, the resulting circuits are significantly easier to test than circuits designed without regard to testability. Wherever fault coverage results are given, layout areas, transistor counts, and critical delays are also given so that the trade-off between a circuit's testability and its area and performance can be fully appreciated.

Examples of our insertion methodologies employ three different BIST schemes: conventional BIST, circular BIST, and the circular self-test path technique. For circular BIST and the circular self-test path technique, special care must be taken when adding the test circuitry to a design. This work explores the problems that can occur, and outlines structural constraints that should be followed to avoid the problems.

The three BIST insertion methodologies introduced complement one another, rather than compete with one another. Although it is desirable to consider testability as early as possible in the design flow, the algorithmic level methodology does not supercede the register transfer and gate level methodologies. We show that each methodology has its own area of application.

METHODOLOGIES FOR BUILT-IN SELF-TEST INSERTION IN VLSI CIRCUITS ACROSS THE DESIGN HIERARCHY

by

JOAN ELIZABETH CARLETTA

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Thesis Advisor: Dr. Christos A. Papachristou

Department of Computer Engineering and Science

CASE WESTERN RESERVE UNIVERSITY

January 1996

CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Joan Elizabeth Carletta

candidate for the Ph.D.

degree.*

(signed) Chris A. Papachristou
(chair)

Steven L. Garrel

Francis Merat

date August 21, 1995

*We also certify that written approval has been obtained for any proprietary material contained therein.

I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

Joan E Carlotta

METHODOLOGIES FOR BUILT-IN SELF-TEST INSERTION
IN VLSI CIRCUITS
ACROSS THE DESIGN HIERARCHY

Abstract

by

JOAN ELIZABETH CARLETTA

Methodologies for built-in self-test (BIST) insertion in VLSI circuits are presented for three different levels of design abstraction. The methodologies are designed to be used during the design flow of application specific integrated circuits (ASICs), which starts at the algorithmic level in the behavioral domain and moves to the register transfer level in the structural domain, and finally to the gate level in the structural domain. At each level, the methodology is based on the use of testability metrics to identify and remove points of low testability in a circuit. By quantifying the properties that make a BIST scheme successful, the testability metrics provide a way to measure test quality implicitly, without resorting to fault simulation, which is both expensive and not available at the higher levels of design abstraction. The testability metrics are computed using a Markov chain model.

Fault coverage curves show that when the BIST insertion methodologies are applied, the resulting circuits are significantly easier to test than circuits designed without regard to testability. Wherever fault coverage results are given, layout areas, transistor counts, and critical delays are also given so that the trade-off between a circuit's testability and its area and performance can be fully appreciated.

Examples of our insertion methodologies employ three different BIST schemes: conventional BIST, circular BIST, and the circular self-test path technique. For circular BIST and the circular self-test path technique, special care must be taken when adding the test circuitry to a design. This work explores the problems that can occur, and outlines structural constraints that should be followed to avoid the problems.

The three BIST insertion methodologies introduced complement one another, rather than compete with one another. Although it is desirable to consider testability as early as possible in the design flow, the algorithmic level methodology does not supercede the register transfer and gate level methodologies. We show that each methodology has its own area of application.

to my parents

Acknowledgments

This work could never have been done without the help and support of a great many people. I owe a great deal to my thesis advisor, Christos Papachristou, for his guidance, technical and otherwise. The members of my proposal and dissertation committees made many helpful suggestions, and I thank them: Daniel Saab, Frank Merat, Steven Garverick, Sheldon Gruber, Mehrdad Nourani, and Warren H. Debany, Jr.

I owe a special debt to Warren Debany of the USAF's Rome Laboratory. Warren first started "teaching me the ropes" of computer engineering almost fifteen years ago, and has been my mentor, in one capacity or another, ever since. I am also grateful to Scott Davidson of AT&T for sharing his perspective on the importance of design-for-testability in "the real world", and for serving as my mentor under the SRC Graduate Fellowship Program. He also made AT&T's GENTEST fault simulator available for my use.

My thanks go to Robert Larsen and Rockwell International for providing the industrial circuit used as an example in Chapter Seven, and for allowing me access to the COMPASS Design Automation suite of VLSI design tools. This support was vital in

obtaining realistic results, without which this work would have been relegated to a pencil and paper exercise. I am also grateful to Daniel Gajski of the University of California at Urvine for providing the BdA high level synthesis system.

This work was supported by the Semiconductor Research Corporation (SRC) under both Grant #DJ527 and the SRC Graduate Fellowship Program. I am grateful to the SRC not only for research support, but also for the help that the SRC was able to give in fostering relationships with members of industry. My early graduate school work was funded by the Air Force Office of Scientific Research under their Laboratory Graduate Fellowship Program.

Fellow students at Case Western Reserve University have been instrumental in my research. I owe special thanks to Mehrdad Nourani, Haidar Harmanani, and Kelly Ockunzzi, who always seem to be around when I need help with a particularly sticky detail. Other students have made valuable contributions as well: Kowen Lai, Wei Zhao, Chih-Hong Fu, Michael Knieser and Mikhail Baklashov.

I owe my most sincere gratitude to my parents, for their constant encouragement, and to my husband, Jinho Lee, for his support, understanding, and patience throughout this endeavor.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ALU	arithmetic logic unit
ASIC	application specific integrated circuit
ATE	automatic test equipment
ATPG	automatic test pattern generation
BdA	Behavioral Design Assistant
BILBO	built-in logic block observation
BIST	built-in self-test
cBILBO	concurrent built-in logic block observation register
CUT	circuit under test
DFT	design-for-testability
MISR	multiple input shift register
PI	primary input
PO	primary output
RTL	register transfer level
SCOAP	Sandia controllability / observability analysis program
SYNTEST	system for synthesis with testability
TPGR	test pattern generation register
VHDL	VHSIC hardware description language
VHSIC	very high speed integrated circuit
VLSI	very large scale integration
XOR	exclusive OR

Chapter 1 *Introduction*

As very large scale integration (VLSI) technology has progressed, and it has become possible to put more and more transistors on a single integrated circuit, testing a circuit to see whether it is functioning properly has become an increasingly difficult task. A variety of techniques for alleviating the testing problem have been developed. This dissertation focuses on one increasingly popular technique, built-in self-test (BIST). The main idea behind BIST is to put everything that is needed to test the circuit directly on the chip, so that the circuit is capable of testing itself. The main thrust of the dissertation is the development of methodologies for automatic insertion of BIST into VLSI application specific integrated circuits (ASICs); given a circuit, we would like to automatically modify the circuit, adding the circuitry that is necessary to perform a built-in self-test. As we do this, we must keep in mind that the addition of circuitry to the chip, while making the chip easier to test, can also have the ill effects of making the chip larger in area (and therefore higher in cost) and slower in speed.

Before we can devise methodologies for BIST insertion, we must have a clear understanding of how ASICs are designed. Contemporary ASIC design is a process consist-

ing of many steps, during which a design flows from one level of design abstraction to another. Typically, a design begins as an "algorithmic" description, almost like a computer program, that specifies the function that the circuit should perform. At this level, the design is a single "black box"; we know what the circuit should do, but not how it should do it. As the design moves from level to level, synthesis tools are used to fill in more and more of the details of how the circuit should be implemented. Early on, we may view the circuit as an interconnection of complex hardware units like adders and multipliers (each still a "black box"); later, the circuit will become a netlist of gates, and ultimately it will be expressed in terms of layout.

Built-in self-test insertion can be done at any point during the ASIC design flow. Generally speaking, it is best to begin thinking about testing early on in the design flow, since how a circuit is designed has a profound impact on how easy it is to test. Once low level decisions about a circuit implementation have been made, it may be too late to easily incorporate testability; making modifications for the sake of testability may require major changes in the design. However, as we shall see later, there are times when we can not begin testability consideration at the highest levels of design abstraction. Therefore, there is need for BIST insertion methodologies that operate over the whole range of design abstraction levels.

1.1 Problem Definition

The goal of this work is the development of methodologies for automatic BIST insertion, in coordination with design synthesis tools. In this work, we describe three separate methodologies for BIST insertion, operating at three different levels of design abstraction, and therefore at three different points in the ASIC design flow. The first works with logic level design descriptions. The second, at a higher level, works with register transfer level descriptions. The third works at the highest level of the ASIC design flow; it operates on algorithmic level design descriptions. Despite differences in design level, the methodologies have many similarities. All take as input a design description, and add BIST features to that design description so that when the description is synthesized to create a physical implementation, that physical circuit can be easily tested. Furthermore, all three methodologies keep in mind four goals:

- to minimize the effect of the BIST insertion on the overall area.
- to minimize the effect of the BIST insertion on the system performance.
- to maximize the fault coverage that can be obtained during test.
- to minimize the test time required to obtain that level of fault coverage.

All use a unified set of testability metrics to quantify the testability of the original circuit and identify trouble spots in the design. These metrics provide a mechanism for trading off area overhead and system performance against obtainable fault coverage and required test time.

We now briefly describe the approach for each of the three methodologies.

1.1.1 Logic level BIST insertion

The first methodology operates on logic level sequential circuits. It starts by assuming a minimal BIST paradigm, in which the primary inputs of a circuit are driven by a test pattern generation register (TPGR) that provides stimuli for the circuit, and the primary outputs are fed into a multiple input shift register (MISR) that analyzes the responses of the circuit to the stimuli. Thus, the primary inputs are assumed controllable, and the primary outputs are assumed observable. It then computes the testability metrics for the internal signals of the circuit; signals with low testability metrics designate possible trouble spots when testing the physical circuit. A two step approach first improves the testability metrics for the flip-flops, so that each combinational logic block receives high quality test patterns, and then improves the testability metrics of internal signals. Two different kinds of insertion, multiplexer-based test point insertion and circular BIST insertion, are used. The output of the methodology is a logic level circuit with added circuitry that is much more easily tested than the original.

1.1.2 Register transfer level BIST insertion

Our second methodology operates on design descriptions at the register transfer level. At this level, circuits are described as interconnections of arithmetic logic units (ALUs) and registers. Like our logic level BIST insertion methodology, our register transfer level BIST insertion methodology begins with a circuit with a minimal amount of BIST; the only BIST added to the circuit is that necessary to drive the primary input registers of the circuit with test patterns, and to compact the responses from

the primary output registers. The methodology then computes testability metrics for the signals of the circuit. Here, the signals are the width of the words in the datapath. The metrics values are used to guide the BIST insertion by finding potential testability problems in the datapath; thus, internal registers of the datapath are replaced with BIST registers only when necessary. The output of the methodology is a modified version of the register transfer level circuit, with added circuitry to make the design easier to test.

1.1.3 Algorithmic level BIST insertion

The goal of our work at the algorithmic level[1] is to develop a methodology for BIST insertion that achieves high fault coverage for both the datapath and the controller created during the ASIC design flow. The methodology is essentially a pre-synthesis process that is meant to be independent of the high level synthesis tool used. Thus, our algorithmic level BIST insertion methodology takes an algorithmic level design description as input, and modifies it to create another algorithmic level description such that when the modified version is synthesized using the normal ASIC design flow, the result is a physical circuit that is easily tested using a simple BIST scheme.

1. Some researchers prefer the term "behavioral level" to the term "algorithmic level" used here.

1.2 Organization of Dissertation

The motivation for this work lies in BIST insertion, which is the process of adding BIST features to a circuit. We consider a number of different BIST methodologies, including conventional BIST, circular BIST, and the circular self-test path technique. Chapter Two provides background by defining some fundamental aspects of design-for-testability. Chapter Three is a survey of recent related research in design-for-testability, especially testability insertion; we cover insertion for both partial scan and BIST because although the two are different applications, much of the underlying motivation is the same. Chapter Four presents mathematical definitions for the testability metrics that form the cornerstone of all our BIST insertion procedures; these metrics are used to pinpoint areas of a design that have testability problems. Chapter Five describes a Markov model that is used to compute the testability metrics.

Regardless of the level of design abstraction at which a BIST insertion procedure operates, the first step in insertion is that of *test point selection*, or deciding where to place test elements (whether registers or flip-flops) within a circuit. For the circular self-test path technique, insertion requires a second step, that of *circular self-test path formation*, in which the selected registers are chained together in a particular *order*, each with a particular *orientation* or shift direction. This second step results in some difficulties unique to the circular self-test path technique. Chapter Six describes these difficulties, and shows how they can be avoided by placing constraints on the structure of the circular self-test path.

The next three chapters provide specifics of our BIST insertion procedures at three different levels of design abstraction. Chapter Seven demonstrates our logic level insertion methodology, using a submodule of an industrial design as both motivation and example. Chapter Eight discusses a similar BIST insertion methodology at the register transfer level, using examples to demonstrate how the metrics facilitate test point selection. Chapter Nine moves BIST insertion into the behavioral domain by considering testability at the algorithmic level. Finally, Chapter Ten presents conclusions, and points towards future research.

Chapter 2 *Fundamentals*

This chapter provides background material preliminary to the main work of this dissertation, and may be safely skipped by readers already acquainted with the design of application specific integrated circuits (ASICs) and design-for-testability. The chapter begins in Section 2.1 with a formal description of the levels of abstraction at which a design can be described. A typical ASIC design flow, from concept to silicon, is presented in Section 2.2. Section 2.3 describes the importance of *design-for-testability (DFT)*, which is the consideration of testability in the course of the design flow. There are the two basic approaches to test, and *deterministic* and *pseudorandom*; Section 2.4 compares them, outlining the advantages and disadvantages of each. Design-for-testability methodologies specific to each approach are introduced; these include both *scan* methodologies for deterministic test in Section 2.5 and *built-in self-test* methodologies for pseudorandom test in Section 2.6. Section 2.7 describes how design-for-testability can be integrated with the design flow. Section 2.8 introduces two key components of testability, *controllability* and *observability*. Section 2.9 is a summary.

2.1 Levels of Design

The levels of abstraction at which a design can be specified are illustrated in Figure 2-1, a Y-chart introduced in [GaKu83]. The three axes represent the three domains in

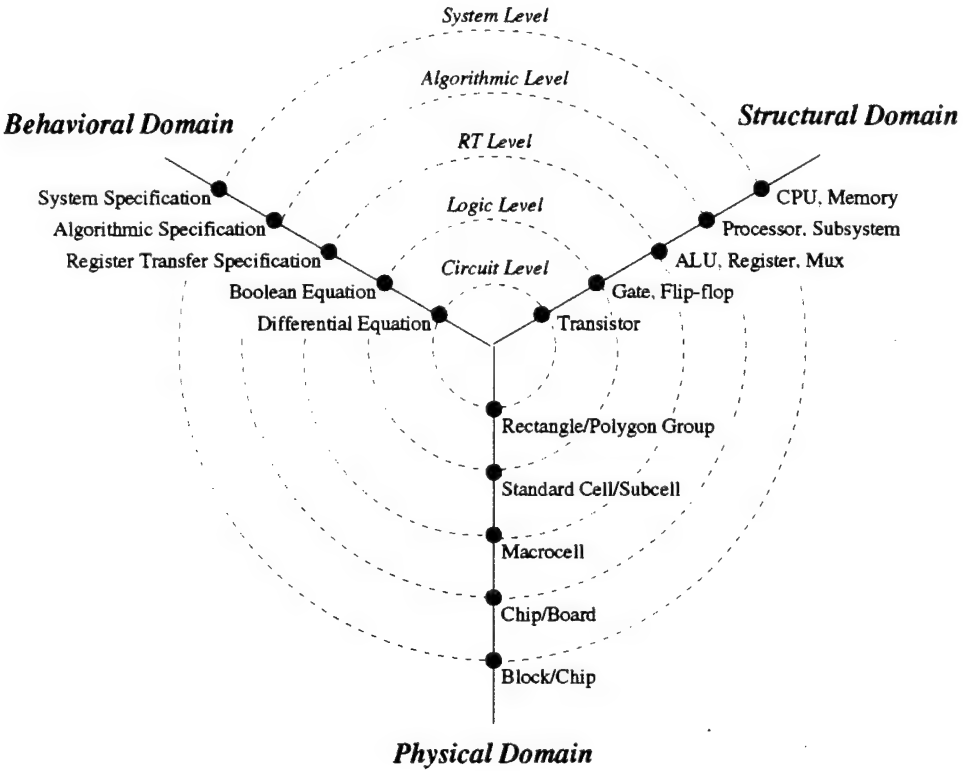


Figure 2-1. Levels of design abstraction.

which a design can be specified. The *behavioral domain* views a design in terms of its function or input-output relationship, without any notion of physical implementation. The *structural domain* views a design as a netlist of interconnected components, where each component is a "black box" element described behaviorally. The *physical domain* views a design in terms of its physical or geometric properties. In each

domain, a range of levels of abstraction are possible, with the more abstract levels at the periphery of the chart. The points on the axes are labelled with a typical view of the design at the given level and in the given domain. For example, from the Y-chart we can see that a register transfer level (RTL) description of a design in the structural domain consists of an interconnection of arithmetic logic units (ALUs), registers, and multiplexers. If we described this same design at the circuit level in the physical domain, it would consist of a number of rectangles and polygons in polysilicon, metal, and other materials used in integrated circuits.

2.2 *ASIC Design Flow*

An ASIC design flow starts with a design description at the algorithmic level in the behavioral domain, and traverses the Y-chart to create a complete physical implementation for the design. A typical path taken is shown in Figure 2-2. The tools used to perform the traversal are as follows:

A. *High Level Synthesis* takes as input a design described at the algorithmic level in the behavioral domain. This input is an algorithm describing the function that the final circuit is intended to perform. The output of high level synthesis is a register transfer level description in the structural domain. This description consists of two parts. The first part is a register transfer level datapath capable of performing the intended function. This datapath is described as an interconnection of arithmetic logic units, registers, and multiplexers; each of these interconnected com-

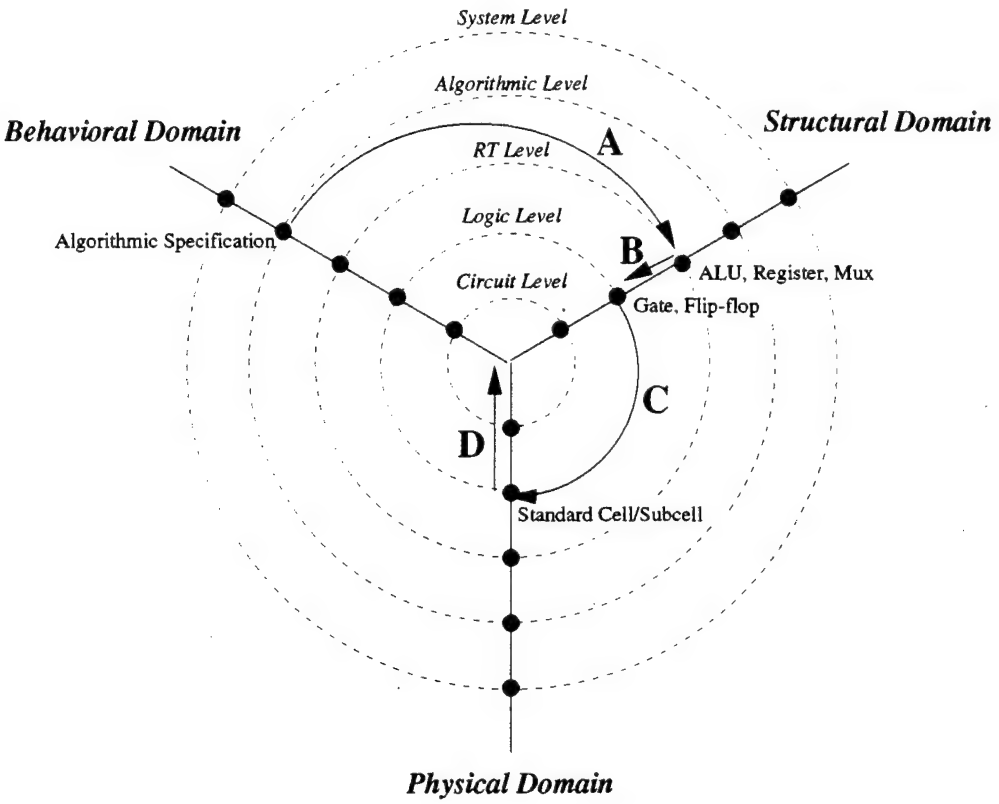


Figure 2-2. Typical ASIC design flow in terms of the Y-chart of Figure 2-1.

ponents is a "black box" described functionally. The second part is a control flow, usually in the form of a state diagram, that describes the way in which the datapath must be controlled in order for it to perform the intended function.

B. *Logic Level Synthesis* takes the register transfer level datapath and transforms it into a logic level implementation in the structural domain. It does this by filling in the "black box" functional description of each component in the datapath with a logic level implementation. At this point, the control flow may also be synthesized to the logic level using a finite state machine implementation.

C. *Technology Mapping* takes as input a gate level design description in the structural domain. It moves the design into the physical domain by mapping the gates to standard cells from a given library. Layout is available for each standard cell.

D. *Placement and Routing* take as input the standard cell implementation of the design in the physical domain. The standard cells are physically situated, and the interconnections between the cells are routed. The result is a layout of the overall design that is ready for fabrication.

2.3 *Importance of Design-for-Testability*

Traditionally, design and test have been considered two separate processes; one group of engineers completed the design of the circuit, and a separate group was responsible for building and testing the circuit. However, as advances in VLSI technology have made higher and higher package densities possible, this approach has become less and less feasible. Testability problems are caused primarily by an inability either to control a signal embedded within a circuit to some required value, or to observe the value of an embedded signal. As package density increases, the number of gates per pin increases. Since the internal parts of the chip can be controlled and observed only indirectly through the pins, this makes testing the chip more difficult. As a result, testability decreases dramatically, and the effort spent in testing a chip becomes a significant portion of the overall cost of getting a new design to market.

Design-for-testability (DFT), or the consideration of testability during the design flow, was introduced in the 1970s as a remedy to decreasing testability. At the beginning, design-for-testability involved designing a circuit so that it had certain general properties that made it easier to test; two such properties are that the design contain no asyn-

chronous logic and that it be easy to initialize to a known state [BaMS87]. Later, more structured DFT techniques were developed. Because the best choice of DFT technique depends on the test approach used, we now digress to introduce the two main approaches to test before coming back to the specifics of design-for-testability. When we return to design-for-testability, we will describe specific DFT techniques appropriate to each of the approaches.

2.4 Approaches to Test

The basic principle of test is to apply stimuli, or *test patterns*, to the circuit under test, and to analyze the responses of the circuit to the test patterns to determine whether the circuit is responding as expected. Test methods can be divided into two basic approaches, depending on the type of test pattern used. *Deterministic test* applies a set of test patterns that are tailor-made for the circuit in question, while *pseudorandom test* applies pseudorandomly generated test patterns. Each approach has advantages and disadvantages. Determining which test patterns to include in a deterministic test is generally computationally intensive, and requires an in-depth analysis of the circuit. This computation of test patterns is usually a one-time cost; however, if even small changes are made to the circuit design, the deterministic test may no longer be valid, and so re-computation of the test patterns may be necessary. In contrast, computing the test patterns for a pseudorandom test is simple, and the test patterns need not be changed when the design is changed. Since the test patterns in a deterministic test are

tailor-made for the circuit under test, they are of high quality; a small number of deterministic patterns can usually catch a large percentage of the faults of the circuit. With pseudorandom test, the patterns may not be of as high quality, so a greater number of patterns may be required.

Deterministic and pseudorandom test also differ in the ways that the test patterns are typically generated during the course of the test. In general, it takes a large amount of hardware to generate a deterministic test; for this reason, the test pattern generation is usually done off-chip, and is the responsibility of automatic test equipment (ATE). In contrast, pseudorandom test patterns can be generated very simply with a minimal amount of hardware. This makes pseudorandom test amenable to *built-in self-test (BIST) techniques* [AgKS93a] [AgKS93b], for which all circuitry needed to test the circuit is placed directly on the chip. A number of different BIST methodologies exist, and will be described later in this chapter. All have in common the addition of test circuitry to the chip to provide test patterns to the circuitry of the chip and to analyze the response of the circuitry of the chip to the test patterns. BIST techniques eliminate the need for expensive ATE; using BIST, the only input that a chip requires from its environment in the course of the test is an indication of when the test should begin. Similarly, the chip gives only limited output reporting the results of the test.

Because all BIST circuitry is on-chip, test can be done at normal circuit speed; in contrast, with deterministic tests using ATE, test must usually be done at a slower-than-normal rate. This means that BIST techniques have the capability to detect timing-

related problems that an off-chip deterministic test may miss. Another advantage of having all test circuitry directly on-chip is that the chip need not be placed in a special environment (i.e., into the socket of an ATE) during test; this means that BIST techniques are useful not only for chip level test, but also for *board level test*. Since the interaction of a chip with its environment during test is minimal, it is relatively simple to place a test controller on a board that asks each chip on the board to test itself.

One disadvantage of BIST techniques is the area required on chip for the test circuitry. The addition of test circuitry can also cause some degradation in system performance, even when the chip is operating in normal (non-test) mode. Thus, a key problem that must be solved in order to make BIST practical is how to add BIST features to a circuit while having a minimal impact on overall area and performance.

2.5 *Scan Methodologies for Deterministic Test*

The problem of test pattern generation has been fully solved only for combinational logic. Because generating deterministic test patterns for sequential circuits can be prohibitively expensive, especially if those circuits have a high degree of feedback, *scan methodologies* are commonly used to simplify the problem [AbBF90] [BaMS87] [John89]. Essentially, scan serves to make the internal flip-flops of the circuit more directly controllable and observable; this is accomplished by adding an operating mode to the circuit in which the flip-flops form a simple shift register or *scan chain*.

For the *full scan methodology*, all internal flip-flops are made accessible by placing them in the scan chain. Since all flip-flops are now accessible, the problem of generating test patterns for the circuit degenerates to that of generating test patterns for the combinational logic of the circuit. A specific test pattern is applied to the combinational logic by placing the scan chain in shift mode and shifting in the test pattern. Then, the scan chain is placed in normal mode, so that the circuit operates according to its designed function; when the circuit is clocked in normal mode, the test pattern is applied to the logic. At this point, the response of the combinational logic to the test pattern is clocked into the scan chain; by putting the scan chain back into shift mode, the response can be shifted out and observed directly.

The *partial scan methodology* improves on the high hardware overhead of full scan by placing only a subset of the flip-flops of the circuit in the scan chain. A variety of techniques exist for deciding which of the flip-flops should be placed in the scan chain; some will be described in Chapter Three, which contains a survey of related research.

2.6 BIST Methodologies for Pseudorandom Test

All BIST methodologies are based on the use of on-chip test circuitry to provide test patterns to the circuitry of the chip and to analyze the response of the circuitry of the chip to the test patterns; the methodologies vary only in how the test pattern generators

and response analyzers are implemented. We now describe several pseudorandom BIST methodologies of special interest; more complete treatments can be found in [AbBF90] and [BaMS87].

2.6.1 Conventional BIST

The conventional BIST methodology uses linear feedback shift registers, also called *test pattern generation registers* (TPGRs), to provide the test patterns. Figure 2-3 shows a typical test pattern generation register. The register itself is a deterministic

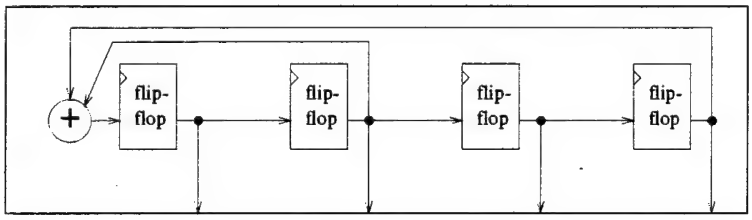


Figure 2-3. A test pattern generation register (TPGR).

system; however, as it is clocked, its states over time satisfy some of the same properties as uniformly distributed random numbers, and so the TPGR is said to provide *pseudorandom* patterns. Conventional BIST also makes use of *multiple input shift registers* (MISRs) as shown in Figure 2-4. Rather than checking the response of the circuit to each test pattern individually, conventional BIST uses an MISR to compress the responses over time into a single *signature* that can be checked at the end of the test session, after all test patterns have been applied. Since the compression of circuit

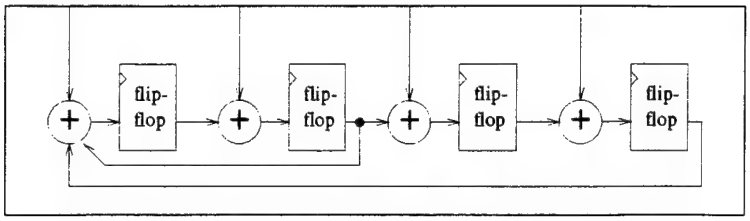
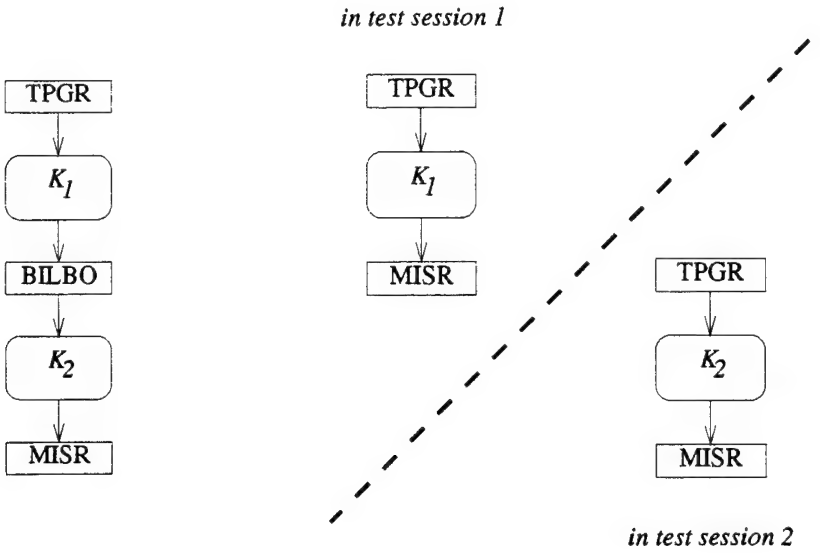


Figure 2-4. A multiple input shift register (MISR).

responses into a signature involves some loss of information, getting the expected fault-free signature is not a guarantee that no fault exists in the circuit; it is possible for a fault effect to be *masked.* The probability that a fault effect is masked is referred to as the *aliasing probability.*

When BIST is added to a circuit, the circuit is usually partitioned into manageable pieces called *kernels*, and BIST registers are placed at the inputs and outputs of each kernel. For example, in a register transfer level (RTL) datapath, the main focus may be on testing the arithmetic logic units (ALUs), and so each ALU may be in a kernel by itself, with TPGRs at the inputs to the ALU, and an MISR at the output of the ALU. If some signal is at the input of one kernel K_1 and the output of another kernel K_2, a third type of test register, the *built-in logic block observation* (BILBO) register, may be used [KoMZ79]. This is a "double duty" register that can act as either a test pattern generation register or a multiple input shift register, depending on its mode of operation at a given time. Since the BILBO can not generate test patterns and compress circuit responses *simultaneously*, kernels K_1 and K_2 must be tested in separate test sessions. This is shown in Figure 2-5; during the test session for K_1, the BILBO acts as an

MISR, compacting the responses of kernel K_1, and during the test session for K_2, the BILBO acts as an TPGR, providing test patterns for kernel K_2.



(a) structure of circuit. (b) operation of BILBO during the test.

Figure 2-5. The use of a BILBO to test two adjacent kernels.

Once a circuit has been partitioned appropriately into kernels, and test registers have been added, test scheduling must be done; this step looks at the requirements on the BILBOs of the circuit, and determines which kernels may be tested in parallel without conflict. Based on this, the kernels are divided up into test sessions. A single test session sets the BILBOs of the circuit to specific operational modes, and then uses those modes to test the kernels assigned to it. Note that, in general, conventional BIST requires a complex test controller; the test controller must be able to move from test session to test session at the appropriate times by sending the proper operational mode controls to the BILBOs.

To avoid the complicated problem of test scheduling, another kind of test register, the *concurrent built-in logic block observation register (cBILBO)*, can be used [WaMc86]. The cBILBO register is like a BILBO, except that it can generate test patterns and compact test responses *simultaneously*; thus, it eliminates the need to test kernels separately. Despite this advantage, the cBILBO is seldom used because of its large area; it is essentially as large as a TPGR and an MISR combined.

2.6.2 MISR-based BIST

MISR-based BIST attempts both to reduce the area overhead required for conventional BIST and to eliminate the need to test kernels separately. The basic scheme is shown in Figure 2-6 for two kernels. A single test session is used to test both K_1 and K_2; during that test session, the MISR between the kernels compacts the responses of K_1, and the partial signatures created in the MISR are used as test patterns for K_2. The main disadvantage of MISR-based BIST is that the partial signatures of K_1 may not be good quality test patterns for K_2.

2.6.3 Circular BIST and circular self-test path

Circular built-in self-test and the circular self-test path technique have recently been proposed as lower cost alternatives to conventional BIST [KrPi87] [PiKK92] [Stro88] [POLB88]. Both methodologies make use of a special test register, shown in Figure 2-7, that can simultaneously both generate test patterns and compact responses. There is no feedback within the circular BIST test registers themselves; instead, the feedback

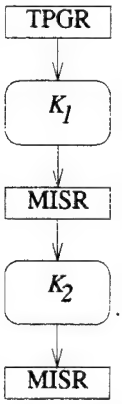


Figure 2-6. An MISR-based BIST scheme for two kernels.

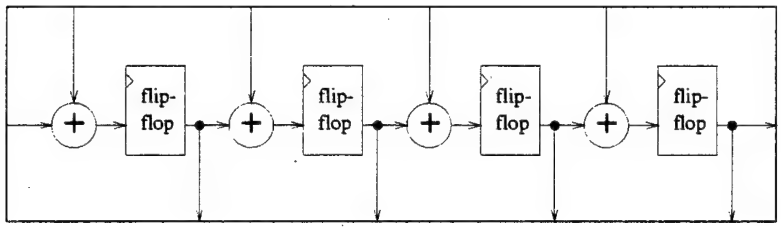


Figure 2-7. A circular BIST test register.

necessary to pseudorandom BIST is created in the way that the test registers are linked together. In the circular self-test path technique, the feedback is formed by linking the test registers together to form one long *circular self-test path*. Figure 2-8 shows an example RTL structure with a circular self-test path added. The circular self-test path is shown with a broken line; for this example, only the primary input and primary output registers are included in the path, although in general any of the registers may be included. For this methodology, the signature is obtained either by watching the stream of bits flowing through a specified point on the path for a specified number of clocks before the end of the test session [Stro88] [PiKK92], or by looking at what

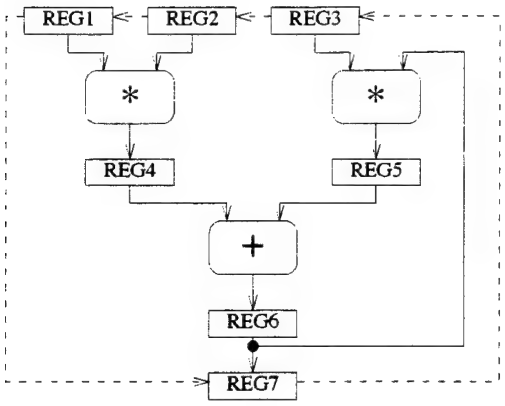


Figure 2-8. An example RTL structure with a circular self-test path.

remains in the test registers at the end of the test session. Note that there is no need to scan out signatures of internal blocks, since all signatures will pass through a primary output register.

For the circular BIST methodology, the test registers are linked together into a long chain; however, the two ends of the chain are not linked to make a circle as in the circular self-test path technique. Instead, the input end of the chain is driven by a TPGR, and the output end of the chain drives an MISR. Figure 2-9 shows the same circuit as Figure 2-8, using the circular BIST methodology instead of a circular self-test path. For this methodology, the signature is obtained from the MISR at the end of the test session.

Both the circular self-test path and circular BIST methodologies offer several advantages over conventional BIST, particularly in regards to hardware overhead. Test control is much simpler for these methodologies than for conventional BIST. First, a

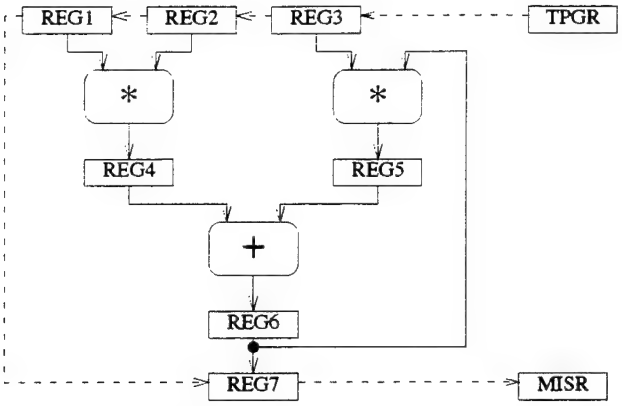


Figure 2-9. An example RTL structure using the circular BIST methodology.

circular BIST register has fewer modes of operation than a BILBO register, so the test controller generates fewer control bits. Secondly, the entire circuit can be tested in one session, so the test controller is not responsible for managing flow of control from one test session to the next, and there is no complicated test scheduling problem to solve.

The main disadvantages of the circular methodologies in comparison to conventional BIST are questions about the quality of the test patterns, which may no longer be truly pseudorandom, and questions about the probability of error masking or aliasing during test response compaction; however, empirical study has shown the circular self-test path technique to be effective at both generating test patterns and compacting responses [PiKK92].

2.7 *Fitting DFT into the Design Flow*

Figure 2-10 shows a typical ASIC design flow from concept to silicon, with the various levels of design abstraction at which the design is described along the way, and the tools that move the design from one level of design abstraction to another. Consider-

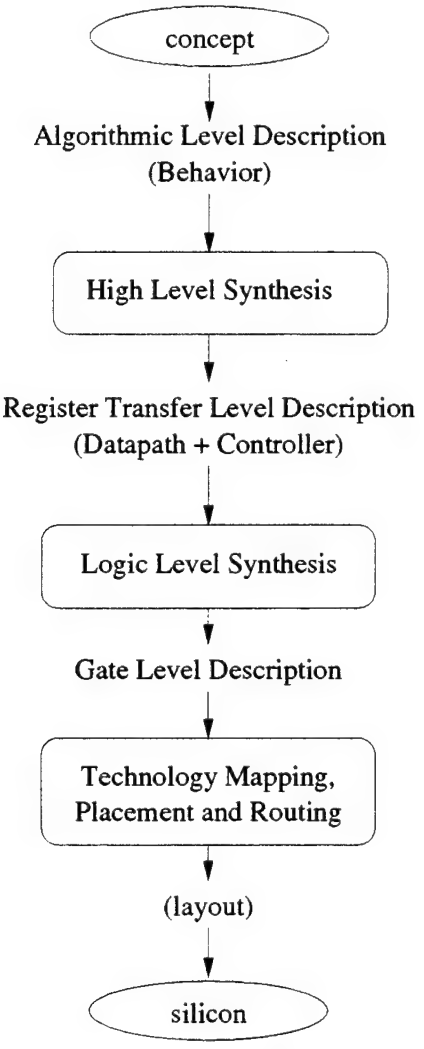


Figure 2-10. The ASIC design flow.

ation of testability can begin at any time in the course of the design flow; however, there are many advantages to considering testability early in the design flow. The most important advantage is that the earlier we consider testability, the more flexibility we have in the design. Design decisions have a profound effect on the testability of a finished circuit; if we wait until a design's physical implementation is nearly complete to start thinking about how to test it, we may have already made design choices that are catastrophic from the point of view of testability. Furthermore, the higher the level of design abstraction, the fewer the number of components that must be considered when analyzing the circuit structure. For example, a circuit with thousands of gates may have only a small number of register transfer level components; this smaller number of components make the analysis problem much more computationally tractable. Additionally, at higher levels of abstraction we may have more information about the circuit; for example, at the algorithmic level we know the function performed by the circuit. This can guide us greatly in designing a test, making the job easier than if all we have is a netlist of gates with no real notion of behavior or how the circuit is meant to be used.

2.8 *Testability Metrics*

In testing a circuit, the bottom line is fault coverage; a circuit is easily testable if a test can be devised that detects a high percentage of the potential faults in the circuit in a reasonable amount of time. However, since fault simulation is expensive, and not

available at levels of design abstraction above the gate level unless synthesis is performed first, it is not feasible to use fault simulation to guide design-for-testability. Instead, *testability metrics* are used to capture the essential aspects of a high quality test indirectly. The two key components of a high quality test are the ability to *control* an internal signal in the circuit to some value that is necessary to sensitize a fault, and the ability to *observe* an internal signal to see whether some fault effect has caused an error in that signal. Testability metrics can be divided into two categories: those that measure controllability, and those that measure observability.

Each signal embedded within a circuit can be thought of as generating test patterns for the part of the circuit driven by the signal. *Controllability* measures whether those test patterns (for that part of the circuit) are of good quality. What metric should be used for controllability depends largely on the test scheme used. For automatic test pattern generation (ATPG)-based test, the issue is one of being able to control the signal in question to a value necessary to sensitize a fault; therefore, a suitable metric should measure how difficult it is to set the signal to the necessary value. We will see several ATPG-based testability metrics in Chapter Three, when we survey related research in design-for-testability. For built-in self-test (BIST)-based test, the issue of controllability is slightly different. We will describe the BIST controllability metrics that we use in this work in Chapter Four.

Observability of a signal measures the degree to which we can tell what is happening at that signal, given that we cannot look at the signal directly, but instead can look only

at the primary outputs of the circuit. In ATPG-based work, metrics for observability may measure the difficulty in sensitizing a path for the fault effect from the signal to a primary output. We will present our own observability metric, suitable for BIST, in Chapter Four.

2.9 *Summary*

This chapter has filled in background information necessary to this research. A typical ASIC design flow was presented in order to clarify the levels of abstraction at which a VLSI design can be described. This concept is central to our work, which develops BIST insertion methodologies applicable at various design levels throughout the ASIC design flow. The two main approaches to test, deterministic and pseudorandom, were presented so that the related research of Chapter Three, which uses both approaches, can be better appreciated. A number of BIST schemes were presented preliminary to their use in later chapters. Finally, the concept of testability metrics, which is central to much of the related research in Chapter Three and is used throughout our own work, is described.

Chapter 3 *Survey of Related Research*

This chapter discusses previous work in design-for-testability, which encompasses techniques to design digital circuits such that they are easy to test once synthesized. We discuss approaches based on both automatic test pattern generation (ATPG) and built-in self-test (BIST), because although the two schemes are different, they share many of the same objectives. We categorize the approaches according to the level of design abstraction to which each approach is applied. We further divide the approaches into testability insertion approaches and synthesis-for-testability approaches.

Testability insertion approaches take a design at a given level of abstraction and modify it, resulting in a more testable design at the same level of abstraction. In contrast, *synthesis-for-testability* approaches modify the synthesis process itself, so that the created circuits are more testable. For example, it has been shown that it is difficult to generate deterministic tests for sequential circuits with feedback loops and large sequential depth. Thus, many ATPG-based design-for-testability approaches focus on the elimination of feedback and the minimization of sequential depth. For testability

insertion approaches, elimination of feedback may entail the addition of test circuitry that breaks any existing feedback during the test; for synthesis-for-testability approaches, the synthesis tool may be modified so that it will not create large feedback loops.

Design-for-testability approaches have another natural division based on the method used. *Structure-based techniques* place *explicit* restrictions on the structure of the circuit in an attempt to improve testability; for example, they may specify that a circuit contain no large feedback loops. *Metrics-based techniques* use *testability metrics* that *implicitly* capture information about testability problems like feedback loops and large sequential depths. Metrics-based testability insertion techniques are typically iterative in nature. First, testability metrics are computed for points throughout a circuit. Next, one point is selected for testability improvement, based on the metrics values, and some modification is made to the circuit to improve the testability at that point. The metrics for the circuit are recomputed, and the process is iterated until all points of the circuit have acceptable testability properties.

	ATPG-based		BIST-based	
logic level testability insertion	♦ Princeton:	[BhJh93]	♦ AT&T ERC:	[POLB88] [LiZB93]
	♦ Iowa:	[PoRe93]	♦ Karlsruhe:	[StWu94]
			❖ this work:	Chapter Seven
register transfer level testability insertion	♦ IMEC:	[SCDM93]	♦ USC:	[LiNB93]
	♦ Linkoping:	[KuPe89] [GuKP94]	♦ UCSD:	[OrHa93]
	♦ Illinois:	[ChWS91a] [ChWS91b] [ChSa93] [ChLP92]	❖ this work:	[CaPa95] Chapter Eight
state table testability insertion	♦ Rutgers:	[ChKA92] [KaCA93] [KaCA95]		
	♦ Iowa:	[PoRe93]		
high level synthesis-for-testability	♦ Princeton:	[LWJA92] [LeJW93a] [LeJW93b] [BhJh94]	♦ CWRU:	[PaCH91] [ChPa91] [HaPa93]
	♦ Montreal:	[JaKa93]	♦ UCSD:	[HaOr94a]
	♦ Wisconsin:	[MuJS94a] [MuJS94b]		[HaOr94b] [VaOr95]
	♦ Linkoping:	[Peng95]		
	♦ Cantabria:	[FeSV94]		
algorithmic level testability insertion	♦ NEC USA:	[DePo94]	❖ this work:	[PaCa95] Chapter Nine
	♦ Illinois:	[ChKS94]		
	♦ UT Austin:	[ViAA92] [VTAA93] [ThVA94]		

Table 3-1. Related research in design-for-testability.

3.1 Logic Level Testability Insertion

We now summarize several projects for testability insertion in logic level circuits.

3.1.1 Princeton University

Bhatia and Jha [BhJh93] decide where to place scan elements for ATPG-based circuits by using controllability and observability metrics that implicitly capture the presence of feedback loops and the sequential depth. This work defines controllability and observability metrics for each latch, primary input, and primary output in the circuit. Primary inputs and scan chain elements have high controllability; for a non-scan latch L, controllability is defined to be the average of the controllabilities of all latches and primary inputs driving the latch L through combinational logic, minus a factor to take into account the increased difficulty in controlling L due to its being a larger sequential depth from the primary inputs. Primary outputs and scan chain elements have high observability; for a non-scan latch L, observability is defined as the average of the observabilities of all latches and primary outputs that latch L drives though combinational logic, minus a sequential depth factor. An iterative approach chooses latches one at a time for inclusion in the scan chain, using the sum of controllability and observability as an overall testability metric.

3.1.2 University of Iowa

Work in [PoRe93] takes a significantly different view of logic level design-for-testability. This work views a logic level circuit as a state machine, and concentrates on modifying that state machine to make the state table strongly connected. A state table is strongly connected if it is possible to traverse from any state to any other state. The state table is modified by adding state transitions to it; this is done by placing some of the state elements in a partial scan chain. The authors describe how strong connectivity is essential in detection of a certain class of faults.

3.1.3 AT&T Engineering Research Center

The work of [POLB88] focuses on circular BIST insertion, using a number of testability metrics to make decisions about which flip-flops should be included in the circular BIST chain. For each flip-flop, four different testability metrics are computed. The first two are based closely on the structure of the cone of combinational logic driving the flip-flop; the first counts how many flip-flops drive that cone of logic, and the second measures the degree of reconvergent fanout within the cone. The third and fourth metrics measure the degree of difficulty in controlling and observing the flip-flop, and are patterned after the SCOAP metrics of [Gold79]. An iterative approach chooses flip-flops one at a time for inclusion in the circular BIST chain, using a weighted average of the four metrics as an overall testability metric.

Work by another group at the AT&T Engineering Research Center looks at a combination of partial scan and BIST [LiZB93]. Some of the flip-flops are selected for inclusion in a scan chain; the selection is structure-based, and concentrates on breaking all cycles and some self-loops. Rather than deriving deterministic test patterns to shift into the scan chain, as is usual for partial scan, the method drives the scan chain with pseudorandom patterns from a TPGR. The output of the scan chain is observed through an MISR. Additional points may be selected for BIST-type test point insertion based on probabilistic controllability and observability metrics. If some signal has low controllability, circuitry is added to bring a new pseudorandom bit to that point during test. If some signal has low observability, an additional primary output is added at that point.

3.1.4 Universities of Karlsruhe and Siegen

Work described in [StWu94] describes a method for BIST insertion that spans the logic and register transfer levels. The method begins by deciding which flip-flops in a logic level circuit need to be test flip-flops; for each test flip-flop, a mode vector is formed showing whether the test flip-flop is used to generate test patterns or collect test responses during each of the test sessions. The method then groups the test flip-flops together to form test registers, using the mode vectors to group test flip-flops that perform the same function at the same time together into a single test register. In this

way, BIST insertion can be done with less overhead than is necessary for a register transfer level approach that makes decisions in terms of whole registers. This method is most applicable to control-dominated structures that contain many self-loops.

3.2 Register Transfer Level Testability Insertion

At the register transfer level, testability insertion decisions are made in terms of whole registers rather than individual flip-flops or latches. The major advantage in doing insertion at the register transfer level is a reduction in computational complexity; since the elements in an RTL circuit analysis are arithmetic logic units, registers, and multiplexers rather than flip-flops or latches and gates, there is an order of magnitude fewer elements to consider at the RTL than at the gate level.

3.2.1 IMEC (Belgium)

The technique of [SCDM93] uses partial scan to break all the feedback loops in a register transfer level circuit. Based on structural analysis of the circuit, the technique uses a depth-first search to enumerate all feedback loops and their possible cuts, and then uses a branch-and-bound algorithm with some pruning to find an optimal solution in terms of the minimal number of latches that must be included in the scan chain in order to break all the loops. The approach does not rely solely on turning existing latches into scan latches to break feedback; it also allows the addition of new scan latches that are transparent in normal mode.

3.2.2 Linkoping University

Like the work in [SCDM93], the work described in [KuPe89] and [GuKP94] uses partial scan to break the feedback loops in register transfer level circuits; however, it uses testability metrics to decide which of the registers involved in the feedback should be made scan registers, and where to place T-cells, which are essentially scan elements that are transparent in normal mode. The work defines testability as a function of four metrics: combinational controllability, combinational observability, sequential controllability, and sequential observability. Primary inputs have high combinational controllability, and the output of an arithmetic logic unit (ALU) has a combinational controllability that is the average of the controllabilities of the inputs to the ALU, times a factor that represents the decrease in controllability due to the functionality of the ALU. Primary outputs have high combinational observability. The input of an ALU has combinational observability that is the product of three factors: the average of the observabilities of the outputs of the ALU; a factor that represents the decrease in observability due to the functionality of the ALU; and the average of the controllabilities of the other inputs to the ALU. The sequential metrics represent the number of clock cycles necessary to control or observe an element. Overall testability is defined as a function of the four metrics, and an iterative approach is used for the actual insertion. First, the feedback loops are broken by selecting one register with poor testability in each loop for insertion. Then, metrics are recomputed, and any other registers with poor testability are fixed, one by one, re-computing the metrics after each insertion.

Finally, T-cell insertion is used to improve the metrics of any lines with poor testability.

3.2.3 University of Illinois

Work presented in [ChWS91a], [ChWS91b], and [ChSa93] selects registers in register transfer level circuits for partial scan, with the goal of easing automatic test pattern generation. It guides the selection based on a testability analysis of a data flow graph representing the circuit's behavior; based on this behavioral analysis, which looks for justification and propagation paths, registers are classified in terms of whether or not they are completely controllable and observable. When a node is not completely controllable, meaning that there is some value that can not be justified at that node, partial scan insertion is done to fix the deficiency.

Another project at the University of Illinois also does partial scan insertion in register transfer level circuits [ChLP92] based on behavioral analysis; however, this work uses a different set of testability metrics. Here, controllability of a node is measured by estimating the minimum and maximum number of instruction cycles required to set that node to a given value. Observability is measured similarly, with estimates of the minimum and maximum number of instruction cycles needed to observe a given value at the node. The first step of insertion is to choose scan registers to break all reconvergent fanout; this is done because the computation method for the metrics does not properly handle reconvergent fanout. Next, an iterative procedure is used to select additional

scan registers, based on the testability metrics and the cost associated with a given selection.

3.2.4 University of Southern California

Work in [LiNB93] describes a method for using BIST to test each arithmetic logic unit (ALU) in a register transfer level circuit without resorting to the kind of full BIST implementation traditionally used, with test registers directly at the inputs and outputs of each ALU. Instead, an analysis looks at possible embedded test environments for each ALU. These embedded test environments are based on the identification of I-paths within the circuit[1]; these are paths that can be used to bring a test pattern from a TPGR or BILBO in some other part of the circuit to the input of the ALU in question, or to bring a test response from the output of the ALU in question to an MISR or BILBO in some other part of the circuit. In general, there are many possible ways to add BIST registers to a circuit so that a suitable test environment exists for each ALU. A branch-and-bound technique is employed to prune the space of possible designs, based on area overhead and test application time for overall designs, which is determined by doing test scheduling and includes an estimate of the test length required for individual ALUs.

1. The "I" in "I-path" stands for "identity", and refers to the fact that a signal does not change in value as it is propagated along the path.

3.2.5 University of California at San Diego

Work in [OrHa93] describes a method for BIST insertion in register transfer level datapaths. The BIST insertion is guided by metrics designed to minimize the test application time for the resulting circuit. These metrics take into account all possible test paths that can be used for testing a module. The metrics also measure the length of a test session for a module when a given test path is used; this length is based on the randomness and transparency properties of the signals at the inputs and outputs of the module as defined in [ChPa91].

3.3 State Table Testability Insertion

This group of research takes a different view of register transfer level design-for-testability; rather than working on datapaths, which are interconnections of ALUs, multiplexers, and registers, these projects view the circuit as a state table or finite state machine. This is a common way to view the controller part of a datapath / controller pair, and general circuits can be viewed in this way as well. The projects described here make modifications in the state table description of the circuit, so that when the circuit is synthesized to the logic level it can be easily tested.

3.3.1 Rutgers University et. al.

Collaborative research done at Rutgers University, NEC USA's C&C Research Laboratories and AT&T Bell Laboratories alters the state table description of a circuit,

embedding a test machine within the design machine [ChKA92] [KaCA93] [KaCA95]. The idea is to add transitions to the state table that are taken only when the state machine is in test mode. The resulting state machine, when synthesized, has two modes of operation, the original design mode and a test mode. These transitions added to the machine to define the test mode are chosen specifically to make testing easier. Transitions in the test machine ensure that for any state of the circuit, there is a short predetermined sequence of inputs that can be used to set the circuit to that state, and a short predetermined sequence of inputs that can be used to propagate the state to the primary outputs.

3.3.2 University of Iowa

In an extension of logic level design-for-testability work described earlier in this chapter, work in [PoRe93] modifies the state table description of a circuit by adding state transitions. This technique focuses on making the state table strongly connected, for reasons already described. In the logic level work, new state transitions were effected using partial scan. At this level, the circuit has not yet been synthesized, so it is possible to simply add the new state transitions directly to the state table.

3.4 *High Level Synthesis-for-Testability*

High level synthesis-for-testability strives to modify the high level synthesis process so that the resulting synthesized circuits are easily testable. [AvMc94] presents an excellent overview of university research efforts in this area.

3.4.1 Case Western Reserve University

[PaCH91] and [HaPa93] describe the use of testability constraints to guide high level synthesis for BIST-based circuits. Their allocation scheme restricts the structure of the synthesized circuit, preventing the formation of certain self-loops that can degrade the quality of BIST tests. In [ChPa91], testability metrics are used to guide the high level synthesis process; these metrics are used to decide when controllable and observable points may be removed, and therefore the metrics allow the system to trade-off hardware overhead with test quality and test application time.

3.4.2 Princeton University

[LWJA92], [LeJW93a] and [LeJW93b] describe work done at Princeton University in high level synthesis of circuits that are easily testable using ATPG. Their structure-based method for register allocation avoids the creation of sequential loops during register binding; when loops can not be completely avoided, the method relies on partial

scan to break any remaining feedback during test. The register allocation also reduces sequential depth and improves the controllability and observability of the registers by binding at least one primary input or output value to each register when possible.

Another ATPG-based project at Princeton, described in [BhJh94], also concentrates on modification of the algorithm that does allocation during high level synthesis. Here, though, the modification concentrates on maintaining test environments for each of the modules in the circuit. A test environment for a module is a set of justification and propagation paths that ensure that the proper test patterns can be justified at the inputs of the module, and that the outputs of the module can be propagated to the primary outputs. By ensuring that each module has a test environment during each iteration of the allocation process, the method ensures that the final synthesized circuit will be testable using ATPG.

3.4.3 Ecole Polytechnique de Montreal

[JaKa93] describes metrics for register transfer level testability analysis. The metrics are designed to be used to guide high level synthesis through incorporation with the cost function used by most high level synthesis systems to choose between competing designs. The metrics, which are based on previous gate level work, measure the difficulty of setting a signal to a specified value, and of propagating a signal error to a primary output. A heuristic for computing the metrics is based on reduced ordered binary decision diagrams.

3.4.4 University of Wisconsin

Work done at the University of Wisconsin [MuJS94a] [MuJS94b] modifies the binding algorithm in an attempt to synthesize a circuit that is more testable using ATPG. Their algorithm, which maps the binding problem onto a minimum-cost network flow problem, tries to avoid bindings that create self-loops and cycles, and that create registers that are not easily controllable and observable. It does this by associating high costs with these undesirable bindings.

3.4.5 Linkoping University

[Peng95] proposes the use of controllability and observability metrics to guide high level synthesis of ATPG-based circuits. The method works by modifying the allocation process to take testability into account. An effort is made to merge nodes with good controllability and bad observability to nodes with good observability and bad controllability; the premise is the merged node will retain the good controllability of the first node and the good observability of the second node.

3.4.6 University of Cantabria

[FeSV94] describes a high level synthesis system for ATPG-based circuits. The system is guided by testability metrics found using sampling while random input vectors are applied to the input algorithmic description. For a single bit, controllability is measured in terms of the probability of forcing the bit to a particular value, and observability is measured in terms of the probability of seeing a change in the bit at a primary

output. The controllability and observability of a signal is defined to be the minimum of the controllabilities and observabilities of the individual bits that make up the signal. An additional sequential depth factor is used to account for difficulty in controlling and observing signals that are far from the primary inputs and outputs. Another metric, the loop-based test cost, is used to quantify the ill effect that the presence of loops can have on the number of scan registers that will be needed to break loops, and the high test pattern generation cost and low fault coverage typical of circuits with loops.

3.4.7 University of California at San Diego

Work in [HaOr94a], [HaOr94b] and [VaOr95] develops a high level synthesis method based on testability metrics that are designed to be helpful in minimizing the test application time required for BIST. The metrics are based on the probabilities that two given modules will be bound together by the synthesis. *Conflict* and *coverage* metrics are indirectly related to the number of test sessions that will be required for the final synthesized circuit; conflict measures the degree of conflict for hardware resources that will occur when an attempt to test the circuit in one test session is made, and coverage measures the accessibility of individual modules. A *correlation* metric measures degradation in test pattern quality due to reconvergent fanout both inherent to the data flow and caused by the binding. The authors show how the metrics can be updated during synthesis, with the initial probabilistic analysis becoming more and more deterministic as binding decisions are made.

3.5 *Algorithmic Level Testability Insertion*

The following group of testability insertion approaches work by modifying the algorithmic level design description prior to high level synthesis.

3.5.1 University of Illinois

Work in [ChKS94] extends previous ATPG-based work at the register transfer level upwards to the algorithmic level. The same behavioral testability metrics used in [ChWS91a], [ChWS91b], and [ChSa93] to select registers for partial scan are used here as a basis for modifying a data flow graph through the insertion of test statements. The test statements, executed when the circuit is placed in test mode, make it easier to find the justification and propagation paths necessary for ATPG.

3.5.2 NEC USA C&C Research Laboratories

Work in [DePo94] proposes modifying the behavioral specification of a design, in the form of a control-data flow graph, prior to high level synthesis so that the synthesized circuit can be easily tested using ATPG with a minimum amount of hardware overhead. Deflection operations, which serve to move operations of the data flow graph without changing the overall functionality of the design, are added with an eye towards reducing the number of scan flip-flops that will eventually be needed to break all loops in the synthesized datapath. Their behavioral transformation approach is specifically designed to be used with their own high level synthesis system, BETS.

3.5.3 University of Texas at Austin

Work in [ViAA92], [VTAA93] and [ThVA94] modifies an algorithmic level behavioral description prior to high level synthesis in an attempt to create a circuit that will be easily testable when synthesized; their method is demonstrated using an ATPG-based test. Their method looks for parts of the input behavior that are incompletely specified; for example, it looks for control modes that are unused. The method then takes advantage of the unspecified parts of the behavior by adding functionality that is conducive for testability; for example, an unused control mode might be used to access a part of the circuit that is ordinarily difficult to access.

3.6 Summary

This chapter has presented a survey of related work recently done in testability insertion and synthesis-for-testability. Our own testability insertion methodologies, described in Chapters Seven, Eight, and Nine for the logic level, register transfer level, and algorithmic level, respectively, are meant to add to this existing work by looking at the same problems with a different emphasis.

Chapter 4 *BIST Testability Metrics*

In Chapter Three, we saw a wide variety of testability metrics used in related research for both automatic test pattern generation- (ATPG-) based and built-in self-test- (BIST-) based design-for-testability. In this chapter, we present a more detailed look at the testability metrics used in our work. As is the case with most testability metrics, our metrics are divided into two categories: those designed to measure controllability, and those designed to measure observability.

4.1 Metrics for Controllability

The basic premise behind built-in self-test is that it is possible to design circuits such that they can be easily tested using pseudorandom test patterns. Therefore, a natural choice for measuring the controllability of a signal is a metric that quantifies the randomness of the signal. *Entropy*, a standard notion from information theory [Papo84], quantifies the uncertainty about the outcome of an event, and is based on the probability distribution for the underlying state space. For example, suppose that a signal X is

one bit wide. Let p_X denote the *1-probability* of signal X, i.e., let p_X denote the probability that signal X takes bit value '1'. Signal X takes bit value '0' with probability $1 - p_X$. If $p_X = 1$, we know that signal X always takes on bit value '1'; similarly, if $p_X = 0$, we know that signal X always takes on bit value '0'. In both of these cases, we are certain about the outcome, and the entropy of the signal is zero. If $p_X = 0.9$, we are quite certain that the signal will take on bit value '1', so the entropy, while not zero, is still quite low. We are least certain about the outcome when $p_X = 0.5$; in this case, the entropy is one, the maximum value for a one-bit signal. For one-bit signals, the full formula for entropy is:

$$I_X = p_X \log \frac{1}{p_X} + (1 - p_X) \log \frac{1}{(1 - p_X)} . \qquad \text{(EQ 4-1)}$$

For signals greater than one bit wide, the principle of entropy is similar. For any signal X, let $|X|$ denote the width of the signal in bits. The current *state* of the signal is the value taken by the signal, and so at a given time signal X may be in any one of the following states: 0, 1, 2, ..., $2^{|X|} - 1$. Let X's state probability distribution be denoted by a row vector $\bar{p}_X$, where

$$\bar{p}_X^T = \begin{bmatrix} p_{X,0} \\ p_{X,1} \\ p_{X,2} \\ \vdots \\ p_{X,2^{|X|}-1} \end{bmatrix} = \begin{bmatrix} \Pr\{\text{signal } X \text{ is in state } 0\} \\ \Pr\{\text{signal } X \text{ is in state } 1\} \\ \Pr\{\text{signal } X \text{ is in state } 2\} \\ \vdots \\ \Pr\{\text{signal } X \text{ is in state } 2^{|X|} - 1\} \end{bmatrix} . \qquad \text{(EQ 4-2)}$$

Here, Pr{E} denotes the probability of the event E. The entropy of the signal X is defined as:

$$I_X = \sum_{i=0}^{2^{|X|}-1} p_{X,i} \log \frac{1}{p_{X,i}}. \quad \text{(EQ 4-3)}$$

Here, entropy ranges from zero, when one event occurs with probability one and the rest occur with probability zero, to |X|, when all events are equally likely to occur.

Following the work of [ThAb89] and [ChPa91], we normalize entropy for use as a controllability metric. We define the *randomness* of a signal X as:

$$\text{MR}(X) = \frac{\text{actual output entropy}}{\text{maximum output entropy}} = \frac{I_X}{|X|}. \quad \text{(EQ 4-4)}$$

As a result, randomness ranges from zero to one regardless of the bit width. The randomness metric can be thought of as a comparison of a signal's effectiveness at generating test patterns with the effectiveness of a "perfect" test pattern generator that generates uniformly distributed random patterns and therefore has the maximum possible entropy. If a signal has randomness zero, the signal generates a single test pattern over and over again; if a signal has randomness one, it generates all possible test patterns with approximately equal frequency.

Another controllability metric that has been used extensively by other researchers for built-in self-test is that of expected state coverage [KrPi87] [PiKK92]. *Expected state coverage* for a signal X is the fraction of all $2^{|X|}$ possible states for that signal that are

expected to be generated or *covered* during a testing session of a given length. The expected state coverage for signal X in N clocks can be written in terms of X's state probability distribution as:

$$\mathrm{ESC}(X, N) = \frac{1}{2^{|X|}} \sum_{i=0}^{2^{|X|}-1} [1 - (1 - p_{X,i})^N], \qquad \text{(EQ 4-5)}$$

as first derived in [DeVH86]. Expected state coverage gives a different perspective from randomness on a signal's effectiveness in generating test patterns. While randomness compares the signal to a perfect test pattern source over the long run, expected state coverage tells us how close the signal comes to generating an exhaustive test during a test session of fixed length. Research by Majumdar and Sastry [SaMa91a] has explored the relationship between state coverage and a more direct measure of test quality, the *fault coverage* obtained for a combinational logic block.

Both randomness and expected state coverage of a signal are based on the state probability distribution of that signal. We now use the circuit of Figure 4-1 to make a point about the calculation of the probability distributions.[1] We restrict ourselves temporarily to trees to simplify the explanation. The controllability of a signal depends only on the circuit that is driving it; this is the part of the circuit between the primary inputs and the signal. This means that if we are considering the controllability of signal Z in Figure 4-1, we need only consider subcircuits A and B, and the operation immediately

1. The symbol ⊗ denotes a generic operation or arithmetic logic unit.

driving Z. We may remove subcircuit C entirely, because it has no effect on the controllability of Z.

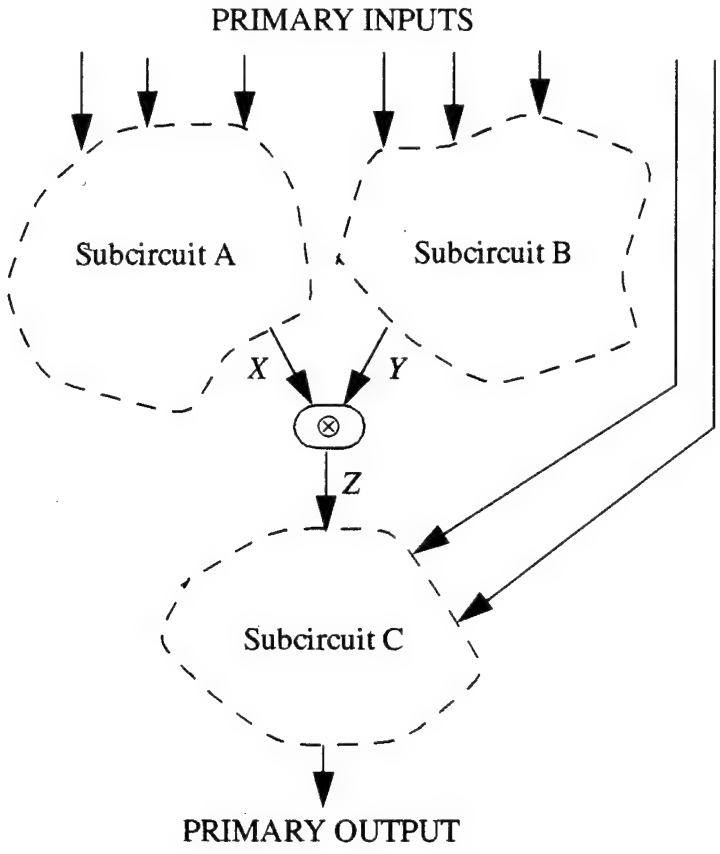


Figure 4-1. Circuit used to illustrate controllability and observability concepts.

Actually, for simple trees, we can write the state probability distribution of signal Z, $\bar{p}_Z$, directly in terms of the state probability distributions of signals X and Y, $\bar{p}_X$ and $\bar{p}_Y$, and the operation ⊗ used to combine X and Y to make Z. For example, suppose that the operation is a modulo-16 adder, so that $Z = X + Y$, and that all of the signals are four bits wide. We can find the probability that signal has value fifteen in terms of all the (X, Y) pairs that sum to fifteen. We have:

$$
\begin{aligned}
p_{Z,\,15} &= \Pr\ \{\text{signal } Z \text{ has value } 15\} \\
&= \Pr\ \{\text{signal } X \text{ has value } 15 \text{ and signal } Y \text{ has value } 0\} \\
&+ \Pr\ \{\text{signal } X \text{ has value } 14 \text{ and signal } Y \text{ has value } 1\} \\
&+ \Pr\ \{\text{signal } X \text{ has value } 13 \text{ and signal } Y \text{ has value } 2\} \\
&+ \ldots \\
&+ \Pr\ \{\text{signal } X \text{ has value } 0 \text{ and signal } Y \text{ has value } 15\}\ .
\end{aligned}
$$

For our tree, signals X and Y are independent, so we can write this as:

$$p_{Z,\,15} = p_{X,\,15} \cdot p_{Y,\,0} + p_{X,\,14} \cdot p_{Y,\,1} + p_{X,\,13} \cdot p_{Y,\,2} + \ldots + p_{X,\,0} \cdot p_{Y,\,15}\ .$$

Other elements of $\bar{p}_Z$ are calculated in a similar way.

It's easy to see how to compute the state probability distributions for an overall tree of operations, using the idea that we can compute a signal Z's state probability distribution in terms of the state probability distributions of the two signals X and Y that immediately drive Z. We start at the primary inputs, which have known state probability distributions; for example, if a primary input signal S is driven with uniformly distributed random patterns, it has a state probability distribution

$$\bar{p}_S = \left[\frac{1}{2^{|S|}} \quad \frac{1}{2^{|S|}} \quad \cdots \quad \frac{1}{2^{|S|}}\right].$$

To compute the state probability distributions of the signals internal to the tree, we work down the levels of the tree one by one until we reach the primary output.

For general circuits, the state probability distribution computation becomes more complicated. In Chapter Five, we will see a Markov model for this computation that takes

reconvergent fanout, direct feedback loops, and indirect feedback loops into account. It also handles the presence of BIST registers. However, even in this general case, the state probability distribution of a signal depends only on the circuitry driving the signal.

4.2 *A Metric for Observability*

For our BIST-based work, we measure the observability of a signal with a new metric that we cal *transparency*. Transparency measures the probability that an arbitrary change in the signal's value can be observed at the primary output. Transparency ranges from a value of 0 for signals that can not be observed even indirectly, to a value of 1 for signals that can be directly observed.

Transparency is inherently more complicated a concept than randomness. To see why, consider what controllability and observability mean in the circuit of Figure 4-1. Again, we begin by restricting ourselves to trees of operations. As previously mentioned, controllability depends only on the circuitry driving a signal; thus, when we consider the controllability of signal Z, we need only consider subcircuits A and B, and the functionality of the operation directly driving signal Z. If observability were a true complement to controllability, the observability of a signal would depend only on that part of the circuit that the signal is driving, i.e., on that part of the circuit between the signal and the primary output. In Figure 4-1, if we were considering the observ-

ability of signal *X*, we would have to consider only subcircuit C and the operation that *X* is immediately driving; we could throw subcircuits A and B away. However, this can not give us a true picture of observability. In actuality, the observability of signal *X* depends heavily on subcircuits A and B. For example, suppose that the operation that *X* is driving is a multiplication. The controllability of the other input to the multiplier, labelled *Y* on Figure 4-1, is a crucial part of the observability of *X*. If signal *Y* always takes value zero, the output of the multiplier will always be zero, regardless of the value of signal *X*; in this case, signal *X* is effectively cut off from ever reaching the primary output. If, however, signal *Y* always takes value one, the multiplier will transmit the signal *X* unchanged to its output, making it easier to propagate *X* to the primary output. Thus, the observability of signal *X* can not be considered without first considering the controllability of signal *Y*. Since the controllability of signal *Y* comes from subcircuit B, we see that we can not remove subcircuit B from consideration.

It turns out that the observability of signal *X* depends on subcircuit A as well. The point behind observability is to measure how easily we can propagate a *change* in signal *X* to the primary output; this change is caused by some fault in the circuit driving signal *X*. The changes that can occur in signal *X* depend on which values signal *X* can take on when no faults are present. This means that if subcircuit A is such that signal *X* is constrained to take on only a certain value during fault-free operation, computing the observability of signal *X* is a different problem than if signal *X* is constrained in some other way.

As a result, the formula for the transparency of signal X will depend not only on the transparency of signal Z, represented by what we will soon define as the state-based transparency of Z, $\bar{t}_Z$, but also on the controllability of signals X and Y, represented by their state probability distributions $\bar{p}_X$ and $\bar{p}_Y$. We now describe the transparency computation in greater detail.

As a tool for evaluating transparency, we define a *state-based transparency vector* $\bar{t}_X$ for each signal X, where each element of the vector is a kind of conditional transparency based on the fault-free state of the signal. We have:

$$\bar{t}_X^T = \begin{bmatrix} t_{X,0} \\ t_{X,1} \\ t_{X,2} \\ \vdots \\ t_{X,2^{|X|}-1} \end{bmatrix} = \begin{bmatrix} \Pr\{X \text{ is transparent} | X \text{ has fault-free state } 0\} \\ \Pr\{X \text{ is transparent} | X \text{ has fault-free state } 1\} \\ \Pr\{X \text{ is transparent} | X \text{ has fault-free state } 2\} \\ \vdots \\ \Pr\{X \text{ is transparent} | X \text{ has fault-free state } 2^{|X|}-1\} \end{bmatrix}.$$

In this definition, signal X is *transparent* if a change in X causes a change in the primary output. The basic idea behind $t_{X,i}$ is this: because of the presence of some fault in the circuit, the value of signal X has been perturbed from its fault-free value of i to some different value. $t_{X,i}$ is the probability that the state error can be propagated from signal X to the primary output. Once we have the state-based transparency vector $\bar{t}_X$ and the state probability distribution vector $\bar{p}_X$, we can write the transparency of signal X as a dot product:

$$\text{MT}(X) = \bar{t}_X \cdot \bar{p}_X^T. \qquad \text{(EQ 4-6)}$$

The state-based transparency vectors are computed in bottom-up fashion, starting with the primary output, and moving signal by signal towards the primary inputs. The primary output is an observable point with perfect transparency, and therefore has $\bar{t} = [1\ 1\ \ldots\ 1]$. We move up one level in the circuit by separating the transparency of a signal X into two components: the probability of propagating a state error through a single operation to the next signal Z in the behavior (see Figure 4-1); and the probability of propagating the state error from that next signal Z to the primary output. Thus, we will write the state-based transparency of X, $\bar{t}_X$, in terms of the state-based transparency of Z, $\bar{t}_Z$. We have:

$$t_{X,i} = \left(\sum_{j=0}^{2^{|Y|}-1} S_{i,j}^{\otimes}\ p_{Y,j} \right) \left(\sum_{j=0}^{2^{|Y|}-1} t_{Z,\, i \otimes j} p_{Y,j} \right). \qquad \text{(EQ 4-7)}$$

This equation shows the two components of the transparency of signal X. The first sum indicates the probability of propagating the state error through the single operation to signal Z. $S^{\otimes}$ is a matrix indicating the *sensitivity* of the operation $\otimes$ to changes in values in X, given a particular fault-free value of X and value of Y. $S_{i,j}^{\otimes}$ indicates the probability that a change in the value of signal X, from a fault-free value of i to some different value i', can be observed at the output of the operation when signal Y has value j. This is the probability that $i \otimes j \neq i' \otimes j$. Although in general the entire sensitivity matrix must be generated, for many practical functions it is not necessary to work with full matrices at all. For example, for addition, all elements of the sensitivity

matrix are equal to one, and the entire first sum of Equation 4-7 collapses simply to the value one.

The second sum in Equation 4-7, based on the transparency of Z, indicates the probability of propagating the state error from signal Z the rest of the way to the primary output. When the fault-free value of signal X is i and the value of signal Y is j, the fault-free value of signal Z is $i \otimes j$; this is the reason that $t_{X,\,i}$ depends on $t_{Z,\,i \otimes j}$.

We have not yet developed a method for transparency computation in general circuits, particularly in those circuits with a high degree of reconvergent fanout. Fortunately, we have found that it is not necessary to handle such circuits. The reason for this is that our test point insertion methodologies always enhance controllability first, before considering observability. Controllability is considered first because enhancing controllability can have an effect on observability. This was the case for our earlier example of a multiplier that had one input tied to zero; recall that this circuit has an observability problem because the zero makes it impossible to propagate the signal on the other input of the multiplier through the multiplier. This circuit also has a controllability problem, since the constant zero is not a high quality test pattern to test the multiplier. If the controllability problem is fixed by modifying the circuit so that the input takes on a wider range of values, the observability problem will be fixed, too. In contrast, enhancing observability has no effect on controllability; regardless of how extra observable points are added to a circuit, the flow of signals through the operations of the circuit remains the same.

Because controllability enhancement is done before observability enhancement, the circuits for which transparency must be computed are not truly general. Controllability enhancement frequently involves the removal of reconvergent fanout, for example, so in practice we do not need to be able to compute transparency for circuits with complex patterns of reconvergence. Simpler patterns of reconvergence are handled with a preprocessing transformation analogous to that used for controllability analysis and introduced in Chapter Five.

One circuit structure that we do need to handle is that of fanout to more than one primary output. Consider the structure of Figure 4-2, where subcircuit A and subcircuit B do not overlap, i.e., where the two paths to the output are independent. We first compute the transparencies of the two signals on the fanout branches, X_A and X_B, by the method already described. Clearly, the transparency of the source signal X is higher than the transparency of either of the branches, since the only requirement for X to be transparent is that we be able to propagate an error on signal X to at least one of the primary outputs. Therefore, there is a choice of path: through subcircuit A, through subcircuit B, or both. We define the transparency of the source as:

$$\mathrm{MT}(X) = \mathrm{MT}(X_A) + \mathrm{MT}(X_B) - \mathrm{MT}(X_A)\,\mathrm{MT}(X_B)\,. \qquad \text{(EQ 4-8)}$$

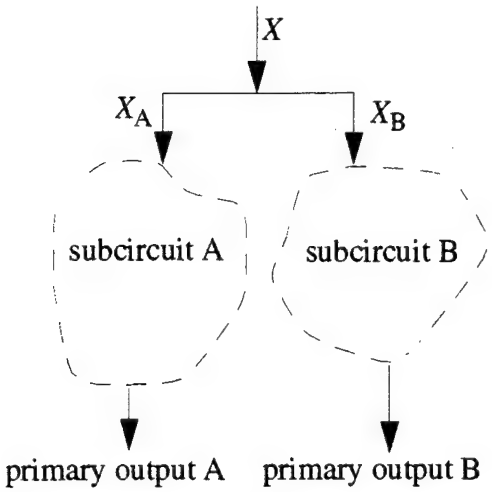


Figure 4-2. A fanout branch.

4.3 *Summary*

This chapter has provided mathematical definitions for the testability metrics used throughout our work. All three of the testability metrics, randomness, expected state coverage, and transparency, are based on the state probability distributions of the signals in the circuit. We will present a Markov model for computing the state probability distributions in Chapter Five.

There are alternatives to using the Markov model to compute the state probability distributions. The probability distributions can be determined empirically with the use of Monte Carlo simulation of the circuit. It is also possible to develop *heuristics* to calculate the testability metrics directly, without relying on the underlying state probability distributions. Heuristics have the potential to be much faster to compute, since they

can work with scalar values rather than the vectors and matrices required for a Markov model approach, or the vectors and long simulation times required for Monte Carlo simulation. Of course, heuristics are inherently less accurate than analytical methods. The development of heuristics for computation of the testability metrics is beyond the scope of this dissertation.

Chapter 5

A Markov Model for BIST Analysis

This chapter develops a Markov model to evaluate the test effectiveness of a circuit with built-in self-test (BIST) features.This model can be applied at various levels of design abstraction; here, we will see its use at the gate level and register transfer level in the structural domain, and at the algorithmic level in the behavioral domain.

The purpose of the Markov model is to provide analytical values for the probability distribution of the state of each signal in the circuit; testability metrics such as entropy-based randomness, expected state coverage, and transparency can be computed from the probability distributions using the formulas given in Chapter Four. By providing a means for BIST analysis and evaluation, the testability metrics allow us to trade area and performance against test effectiveness when doing BIST insertion.

The Markov model described in this chapter is based on previous approaches for BIST analysis at the register transfer level. The Markov model used here provides the following advances over previous models by Chuang and Gupta [ChGu89] and Kim, Ha, and Tront [KiHT88]:

- a preprocessing transformation is used to remove reconvergent fanout from the circuits, allowing the effects of *word-level correlation* on test quality to be accurately modeled.
- an iterative technique is developed so that the model can handle circuits with *indirect feedback*, i.e., circuits in which a register or flip-flop feeds back into itself via one or more intermediate registers or flip-flops.
- the model is extended to include the newer *circular BIST* methodology as well as conventional BIST.

This work also uses the model in a new way, to evaluate testability metrics as an aid to BIST insertion.

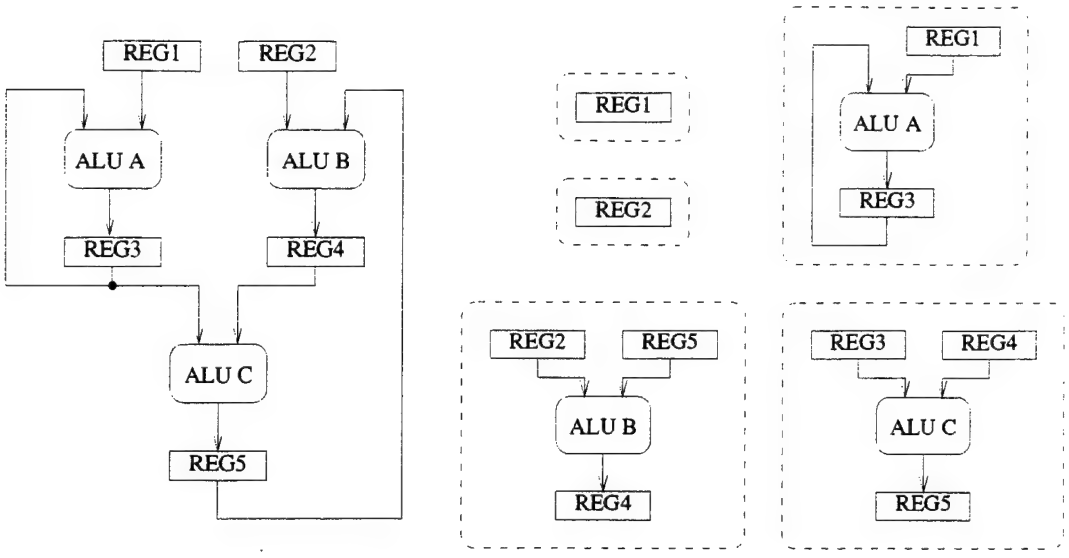
The chapter is organized as follows. Section 5.1 presents the mathematical details of the Markov model for register transfer level (RTL) circuits. Section 5.2 describes a transformation technique for removing reconvergent fanout from an RTL circuit, provides an example transformation, and explains how the preprocessing transformation is used to make the Markov model more powerful. Section 5.3 discusses the computational complexity of the overall analysis for RTL circuits. Section 5.4 shows the same Markov model as applied to gate level circuits; derivations are given to show that the matrix computations necessary at the register transfer level are reduced to scalar equations at the gate level. Section 5.5 describes how the Markov model can be applied at the algorithmic level, in the behavioral domain. Finally, Section 5.6 presents a summary.

5.1 The Markov Model at the RTL

This section describes a Markov model for the analysis of register transfer level (RTL) circuits with BIST features. For the purposes of this section, RTL circuits consist of registers, arithmetic logic units (ALUs) and buses. At this point, we consider only datapaths, and not control logic. First, we describe how the datapath is partitioned before analysis in order to make the problem more computationally tractable. Next, we describe the steps in analyzing a single register of the datapath. Then, we outline the iterative procedure that is used to resolve the analysis when a datapath contains indirect feedback.

5.1.1 Partitioning a RTL datapath for analysis

It is impossible to analyze most RTL datapaths as a whole, since a datapath with even a moderate number of registers has a very large, unmanageable number of system states. As a result, we must *partition* the datapath into smaller pieces for analysis. Note that if we were able to analyze the datapath in one big piece, the analysis would be deterministic; given a current system state and a set of primary input values, there is only one possible next system state. The price we pay for partitioning is that any analysis must now be *probabilistic*, rather than deterministic; since we are looking at only a small section of the datapath at a time, we can not speak with certainty about what is happening in the rest of the datapath.



(a) a simple RTL datapath. (b) the datapath partitioned.

Figure 5-1. An example of partitioning for analysis.

For this application, we'd like to partition the datapath such that each partition contains the information necessary to analyze a single register. The rules for partitioning are simple: create one partition for each register of the datapath, where the register serves as the output register of the partition. The partition should consist of the register, any combinational logic that drives the register, plus the registers that serve as inputs to that combinational logic. The partition for a register is unique. The partitions represent the transfer of data from register to register. Figure 5-1 shows a simple RTL datapath and its partitions. Note that partitioning does not remove feedback loops from the datapath. Direct feedback manifests itself as feedback within a partition, as is the case with register 3 of this example, and is resolved by the Markov model. Larger feedback loops manifest themselves as interdependency among the partitions; for this example, we see that the partitions for registers 4 and 5 are interdependent, since regis-

ter 5 is an input to the partition for register 4, and vice versa. Subsection 5.1.3 explains how this type of feedback is resolved. In Section 5.2, we will see that this simple partitioning scheme is not directly applicable to all datapaths; there, a preprocessing transformation will be presented for use on some datapaths before partitioning. This will enable the correct modeling of a more general class of circuits.

5.1.2 Analyzing a single register

The objective of analyzing a single register is to find the register's state probability distribution $\bar{p}$ as defined in Equation 4-2 on page 47. The steps for finding $\bar{p}$ are described in the procedure of Figure 5-2. The remainder of this subsection describes the steps of the procedure in greater detail.

Procedure *AnalyzeRegister*

Step 1. Find Q, a matrix that describes the register's state transitions when BIST is disabled.

Step 2. Modify Q to compute C, a matrix that describes the register's state transitions when BIST is enabled.

Step 3. Use C to compute $\bar{p}$, a row vector describing the steady-state probability distribution over the register's state space.

Figure 5-2. Procedure for analyzing a single register.

Step 1: Computing the BIST-disabled state transition matrix Q

The first step in analyzing a register X is to compute a matrix describing the register's transitions from state to state during normal (BIST-disabled) operation. Let Q denote this *state transition matrix*. The matrix Q is $2^{|X|}$ by $2^{|X|}$, where $|X|$ is the bit width of the register. Q has elements

$$q_{ij} = \Pr\{\text{next register state is } j \mid \text{current register state is } i\} \quad \text{(EQ 5-1)}$$

for $i,j = 0, 1, ..., 2^{|X|} - 1$. The register's next state is written in terms of the *local* structure of the datapath, more specifically in terms of that part of the datapath contained within the register's partition. The state transition matrix Q depends on the functionality of the combinational logic within the partition, and the probability distributions of any registers that serve as input registers to the partition. The matrix Q also depends on whether there is direct feedback within the partition, i.e., on whether the output register feeds back into itself through the combinational logic of the partition. A register that feeds back directly into itself is said to be *self-adjacent* [HuPe87]. Note that if a register is not self-adjacent, the next state of the register does not depend on the current state, and the equation for q_{ij} is reduced to

$$q_{ij} = \Pr\{\text{next register state is } j\} \quad \text{(EQ 5-2)}$$

for $i,j = 0, 1, ..., 2^{|X|} - 1$. In this special case, all rows of the state transition matrix Q are identical.

Step 2: Modifying Q to compute the BIST transition matrix

Step 2 of the procedure *AnalyzeRegister* modifies Q, the register's state transition matrix for the BIST-disabled case, to compute C, the state transition matrix for the register when BIST is enabled. The elements of C are defined by:

$$c_{ij} = \Pr\{\text{next register state is } j \mid \text{current register state is } i\} \quad \text{(EQ 5-3)}$$

keeping in mind that BIST is enabled. How C is computed depends on the BIST methodology used. For example, suppose that the register is a multiple input signature register (MISR). From the way that MISRs work (see Figure 2-4 on page 18 for a gate level view), we know that the next state of the register is the bitwise XOR of what the next state would be if BIST were disabled and a shifted version of the register's current state, with a modulo-2 sum of the values at the feedback taps shifted into the newly vacated bit. Suppose that the register moves from state i to state k when BIST is disabled; when BIST is enabled under the same conditions, the register will move from state i to state $k \oplus \text{SHL}(i, f(i))$,[1] [2] where $f(i)$ is the value of the feedback bit. Information about what the next state would be if BIST were disabled comes from the matrix Q, computed in Step 1. For MISRs and TPGRs, each element of C can be expressed in terms of Q by:

$$c_{ij} = q_{i,j \oplus \text{SHL}(i,f(i))}. \quad \text{(EQ 5-4)}$$

---

1. $\oplus$ denotes a bitwise XOR.
2. SHL(i, b) denotes a left bit shift of the binary representation for i, with the bit b shifted into the newly vacated least significant bit.

The circular BIST and circular self-test path cases are similar, except that the next state of a register depends also on the bit value being shifted into the register from the circular BIST chain. From the way that a circular BIST test register works (see Figure 2-7 on page 21 for a gate level view), we know that the next state of a register is the bitwise XOR of what the next state would be if BIST were disabled and a shifted version of the register's current state, where a bit from the preceding register on the circular BIST chain is shifted into the vacated bit position. Consider register 7 of the datapath of Figure 5-3 as an example. If BIST were disabled, register 7's next state would be the value coming into register 7 from the data path (i.e., from register 6). Since BIST is enabled, this value is bitwise XORed with a shifted version of the current state; this shifted version is a right shift of the current state, where the newly vacated most significant bit is filled with a value coming from the circular self-test path, namely, the most significant bit of register 1.

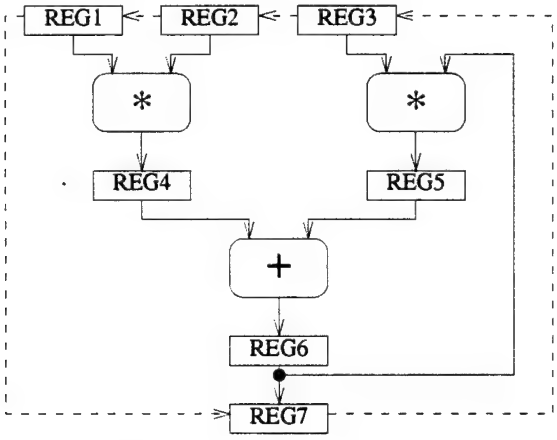


Figure 5-3. An example RTL datapath using the circular self-test path technique.

For the general case, let ρ denote the probability that the bit shifted into the register from the circular BIST chain is a 1. Since the bit comes from either the most or least significant position of the preceding register T (depending on whether that register is shifted to the right or to the left), ρ can be written in terms of T's state probability distribution, where $p_{T,i}$ denotes the probability that register T is in state i (see Equation 4-2 on page 47):

$$\rho = \Pr\{\text{shifted-in bit is a 1}\} = \begin{cases} \sum_{i=2^{|T|-1}}^{2^{|T|}-1} p_{T,i} & \text{if register } T \text{ is left-shifted} \\ \sum_{i=0}^{2^{|T|-1}-1} p_{T,2i+1} & \text{if register } T \text{ is right-shifted.} \end{cases} \quad \text{(EQ 5-5)}$$

Thus, for a circular BIST test register, each element of C can be expressed in terms of Q and ρ by the following:[3]

$$c_{ij} = \begin{cases} \rho q_{i,j \oplus \mathrm{SHL}(i,1)} + (1-\rho) q_{i,j \oplus \mathrm{SHL}(i,0)} & \text{if register } R \text{ is left-shifted} \\ \rho q_{i,j \oplus \mathrm{SHR}(i,1)} + (1-\rho) q_{i,j \oplus \mathrm{SHR}(i,0)} & \text{if register } R \text{ is right-shifted.} \end{cases} \quad \text{(EQ 5-6)}$$

Equation 5-6 shows that each element of C is written in terms of two different elements of Q. In fact, each row of C is a linear combination of two different permutations of the same row of Q.

3. SHR(i, b) denotes a right bit shift of the binary representation for i, with the bit b shifted into the newly vacated most significant bit.

Step 3: Computing the register state's probability distribution

Step 3 of the procedure *AnalyzeRegister* uses C, the register's state transition matrix when BIST is enabled, computed in Step 2, to compute the register's state probability distribution in the steady state. Suppose that $\bar{p}(t)$ denotes the probability distribution at time t. The register state transition matrix C relates $\bar{p}(t+1)$ to $\bar{p}(t)$:

$$\bar{p}(t+1) = \bar{p}(t)\,C \ .$$

It's easy to see that C is a Markov matrix [Stra88]; all of its elements are nonnegative, and each row adds to 1. Like all Markov matrices, C has an eigenvalue equal to 1. This means that the probability distribution will settle down to a steady-state; it is this steady-state probability distribution $\bar{p}$ that we need. We have

$$\bar{p} = \bar{p}C \ . \qquad \text{(EQ 5-7)}$$

From this equation, $\bar{p}$ is the left eigenvector of C corresponding to eigenvalue 1. We can solve the equation for $\bar{p}$ in the following way [Stra88]:

- *Step A*. Compute $C^T - I$.
- *Step B*. Do a QR decomposition of $C^T - I$, $Q^T(C^T - I) = M$.
- *Step C*. Solve the upper triangular system $M\bar{p}^T = 0$ for $\bar{p}$.

It is always the case that the matrix M is rank deficient, so that $\bar{p}$ has a non-trivial (non-zero) solution.

5.1.3 Iterating to analyze the entire datapath

Two steps (Steps 1 and 2) of the procedure *AnalyzeRegister*, given in Figure 5-2, use information about other registers in the datapath, and assume that these other registers have already been analyzed. For some datapaths, it is possible to analyze the registers in such an order that the information needed has already been computed; for these datapaths, each register need be analyzed only one time. However, for datapaths containing indirect feedback, it is impossible to choose such an order. For example, consider the datapath fragment of Figure 5-4, which has indirect feedback. Analysis of register REG5 requires knowledge about register REG4's probability distribution, and vice versa. The situation is similar for any datapath using the circular self-test path technique, since each register in a circular self-test path feeds back into itself indirectly through the path. In order to handle these cases, all registers are assigned arbitrary initial probability distributions, and an iterative process, shown in Figure 5-5, is used to repeatedly analyze registers until the probability distributions have reached steady-state values. In general, there is no guarantee that the iterative process will terminate; however, in our experience, the process does terminate for practical cases, and most datapaths require only a few iterations.

We now turn to a transformation that can be used to process the datapaths before applying the Markov model, making the model applicable to a wider variety of datapaths by enabling it to accurately handle word-level correlation. We describe the trans-

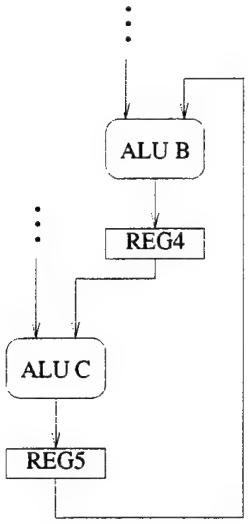


Figure 5-4. An RTL datapath fragment with indirect feedback.

formation on datapaths at the register transfer level, although it is applicable to circuits at the gate and algorithmic levels as well.

5.2 *A Circuit Transformation Technique*

This section describes a technique for transforming a general RTL circuit into a functionally equivalent RTL circuit with no reconvergent fanout. Probabilistic analysis of any kind can be greatly simplified by assuming that there is no correlation among the input registers of a partition, i.e., that one input register's state is independent of every other input register's state. In practice, *word-level correlation* occurs frequently, and is a direct result of fanout in the circuit structure. For example, consider the circuit of Figure 5-6(a). Figure 5-6(b) of the figure shows a direct partitioning of the circuit. When register 6 is analyzed, information about registers 4 and 5 is used. The states of

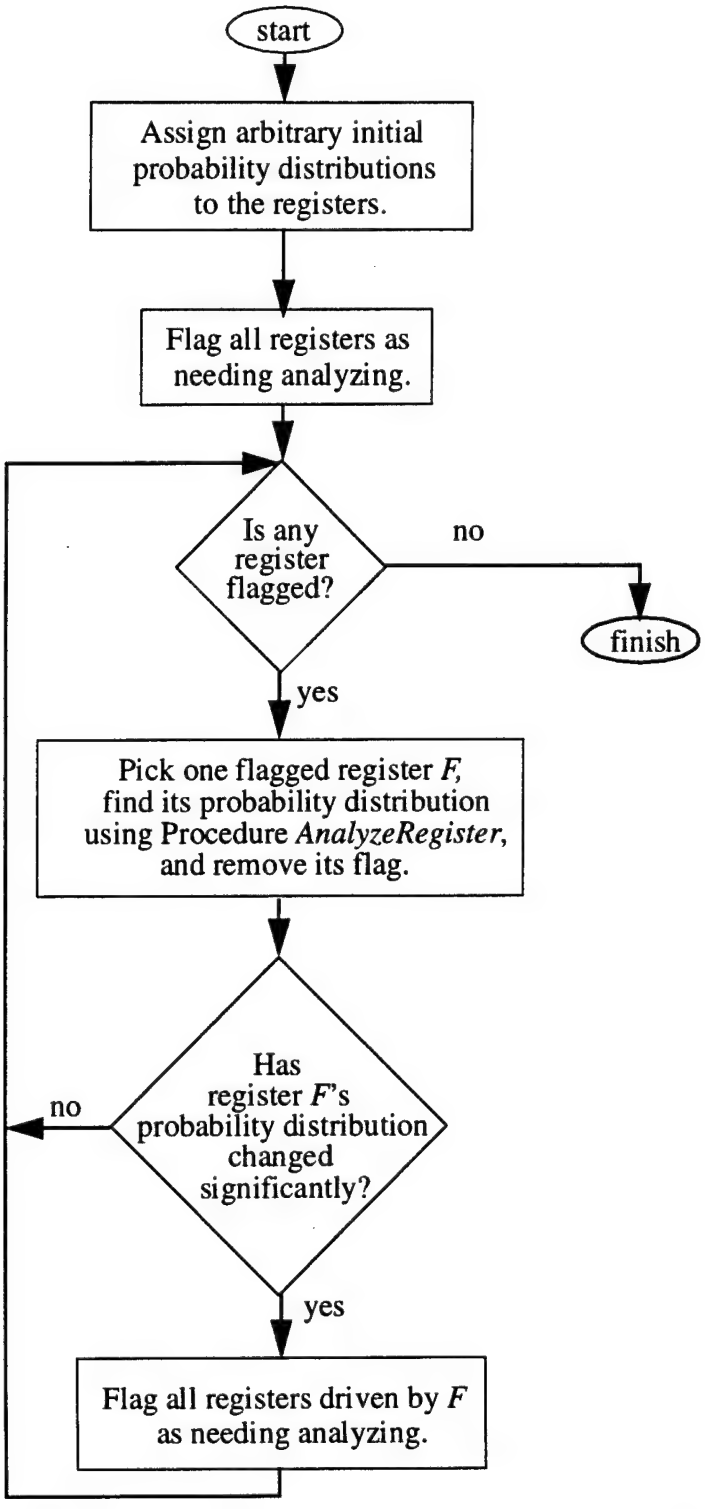
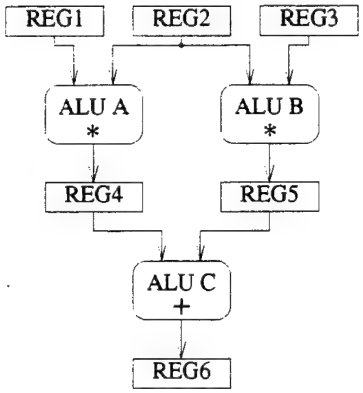
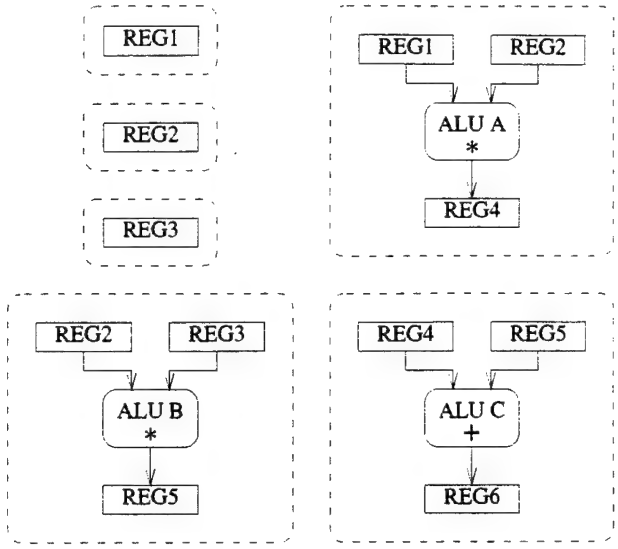


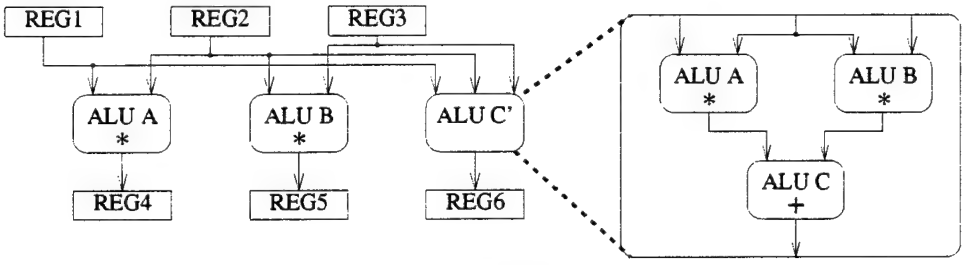
Figure 5-5. Iterative procedure for analyzing an RTL datapath.



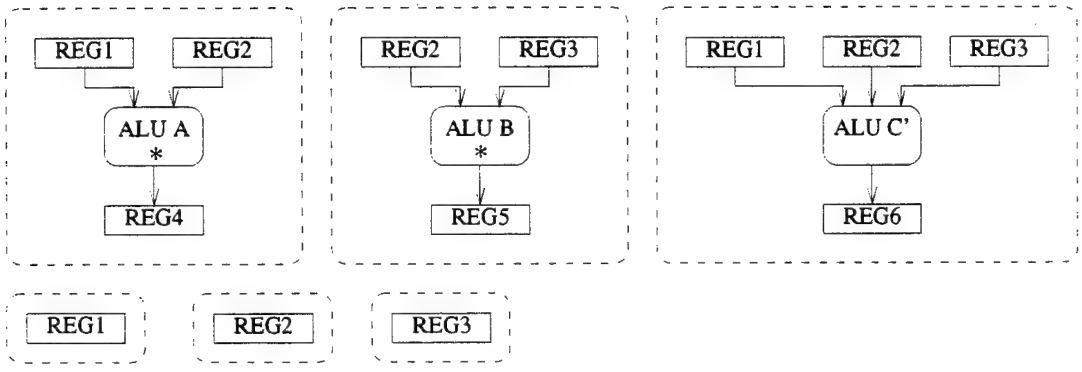
(a) An RTL circuit with reconvergent fanout.



(b) A direct partitioning of the circuit.



(c) The circuit transformed.



(d) A partitioning of the transformed circuit.

Figure 5-6. A simple transformation example.

these two registers are heavily correlated; both are functions of the state of register 2 at the previous clock. Any probabilistic analysis operating directly on the original circuit should take this correlation into account.

Figure 5-6(c) and Figure 5-6(d), which show the transformed circuit and the corresponding partitions, respectively, demonstrate how the transformation technique presented here simplifies the probabilistic analysis of a circuit by enabling us to express a register's state in terms of a set of uncorrelated, *statistically-independent* source registers. The reconvergent fanout is subsumed by the combinational logic, where it can be taken into account easily; the penalty is that we must now work with more complex combinational logic blocks. If desired, a dummy register may be added between CLB *C'* and register 6 to preserve the timing of the circuit by providing an additional clock cycle of delay; for many kinds of probabilistic analysis, including this one, such timing details are not important, and the register can be omitted. For the transformed circuit, the inputs to each partition are independent, and subsequent probabilistic analysis may ignore correlation.

Note that the transformation described in this section does not remove feedback loops from the circuit. Direct feedback loops are taken into account by the Markov model, as previously described in Subsection 5.1.2, and indirect feedback loops are resolved through the iterative process previously described in Subsection 5.1.3.

5.2.1 Preliminary definitions

Correlation between registers is determined by the sources of the registers, and therefore information about the flow of data through the circuit is needed. We say that register T is a *parent* of register R if there is a path from T to R that passes through only combinational logic.[4] The set of registers that are parents to register R is denoted $\mathcal{P}(R)$. Next, we say that register T is a *grandparent* of register R if there is a register X such that T is a parent of X and X is a parent of R. In the circuit of Figure 5-6(a), register 6 has two parents (registers 4 and 5) and three grandparents (registers 1, 2, and 3). Since we must trace the sources driving R from an arbitrary distance up the circuit, we now generalize the concept of parents in the following way: a register T is a *distance-i source* of register R if there is a path from register T to register R that passes through exactly i intermediate registers. Thus, parents are distance-0 sources, grandparents are distance-1 sources, and so on. Note that if a circuit contains feedback, it is possible for a register R to be a distance-i source of itself. The set of distance-i sources of R is denoted $\mathcal{A}_i(R)$.

Two registers *R1* and *R2* are said to have a *common source* if there is a register T and a distance i such that register T is a distance-i source of both *R1* and *R2*. If two registers have a common source, their states are *correlated*, that is, the probability that register *R1* is in a given state i is not statistically independent of the probability that register *R2*

4. This includes the trivial case of a direct register-to-register connection, for which the path from T to R has no intervening logic.

is in a given state j. Again, the purpose of the transformation is to ensure that when we partition a circuit, each register's state is expressed in terms of a set of *uncorrelated* source registers.

5.2.2 Finding the sources of a register

The sources of a given register R can be found using the simple recursion

$$\mathcal{A}_i(R) = ParentsOfRegisterSet(\mathcal{A}_{i-1}(R)),$$

that is, the distance-i sources of R are the parents of the distance-$(i-1)$ sources of R. If the circuit contains no feedback, register R has a finite number of generations of sources; in tracing up the circuit, the primary input registers are eventually reached. However, if R or one of its sources is in a feedback loop, R has an infinite number of generations of sources. Figure 5-7 formalizes an algorithm for finding the sources of a register.

5.2.3 Transforming and partitioning the circuit

An algorithm for transforming and partitioning the circuit is shown in Figure 5-8. Given a register R, it returns information about R's partition in the transformed version of the circuit. The main point here is that we would like *to keep the partitions as local as possible*. At the same time, we may not be able to choose partitions in the straightforward way described in Subsection 5.1.1, since we would like to be sure that the

Algorithm *ParentsOfRegisterSet*
Input: An RTL circuit and a set of designated registers $\mathcal{S}$.
Output: The set of parents of $\mathcal{S}$, $\mathcal{P}(\mathcal{S})$.

Set $\mathcal{P}(\mathcal{S}) = \varnothing$.
for $j = 1$ to $|\mathcal{S}|$; /* iterate through registers in $\mathcal{S}$.*/
  Let R be the j^{th} register in the set $\mathcal{S}$.
  Set $\mathcal{P}(\mathcal{S}) = \mathcal{P}(\mathcal{S}) \cup \mathcal{P}(R)$.
end;

Algorithm *SourcesOfRegister*
Input: An RTL circuit with a designated register T.
Output: The number of generations *gen* of sources of register T, and the sets $\mathcal{A}_i(T)$ for $0 \le i < gen$.

if $(\mathcal{P}(T) = \varnothing)$ do;
  /* Register T has no parents, and therefore no sources. */
  Set *gen* to 0.
  return;
end;

Set $\mathcal{A}_0(T) = \mathcal{P}(T)$.
Initialize *gen* to 1.
while (true) do;
  /* Trace back another level in the circuit. */
  if (*ParentsOfRegisterSet* $(\mathcal{A}_{gen-1}(T)) = \varnothing$) then do;
    return; /* We have traced back to the primary inputs. */
  end;
  else do;
    Set $\mathcal{A}_{gen}(T) =$ *ParentsOfRegisterSet* $(\mathcal{A}_{gen-1}(T))$.
    increment *gen*;
    if $(\mathcal{A}_{gen-1}(T) = \mathcal{A}_j(T)$ for some j, $0 \le j < gen - 1)$ do;
      /* We have traced all the way around a feedback loop. */
      return;
    end;
  end;
end;

Figure 5-7. Algorithm to find the sources of a register.

input registers to the partition have statistically-independent states. The algorithm starts with a straightforward, small partition, and enlarges it one step at a time as necessary until all correlation among the input registers of the partition is eliminated.

As the algorithm proceeds, it must keep track of the expression for the state of the output register in terms of the states of the input registers to the partition. This expression is most easily stored as a rooted binary tree with the internal nodes of the tree representing the operations performed by the combinational logic blocks, and the leaves of the tree representing the registers serving as inputs to those operations. Such a tree is commonly called an *expression tree*. When the expression tree is traversed in depth-first order, the expression is created in infix notation. Let $\mathcal{I}$ denote the set of registers serving as leaves of the tree; these registers are the input registers to the partition.

As an example of the transformation algorithm and the use of the expression tree, consider the circuit of Figure 5-6(a). We would like to find the proper partition for register 6. We start by expressing the state of register 6 as a function of the states of register 6's parents, as shown in Figure 5-10(a). At this point, the set of input registers to our partition is $\mathcal{I} = \{R4, R5\}$. We now ask whether we have written the state of register 6 in terms of a set of *statistically-independent* registers; the answer is no, since registers 4 and 5 have a source in common (register 2 is a distance-0 source of both register 4 and register 5). As a result, we expand our expression tree around registers 4 and 5 using

Algorithm *TransformAndPartition*
Input: An RTL circuit with a designated register R.
Output: Information about register R's partition in a transformed version of the circuit, specifically:
$\mathcal{I}$: the set of input registers for the partition.
$\mathcal{E}$: an expression tree for the state of R written as a function of the states of the registers in $\mathcal{I}$.

if ($\mathcal{P}(R) = \varnothing$) then do;
/* In the special case that R has no parents, return a trivial partition. */
Set $\mathcal{I} = \varnothing$ and $\mathcal{E}$ = NIL.
return;
end;

/* Let $\mathcal{I}$ designate the input registers to the partition; start by initializing the expression tree for R's state so that it expresses R's state as a function of the states of R's parents. */
Set $\mathcal{E}$ to a single leaf node, register R.
$\mathcal{E}$ = *ExpandExpressionTree* ($\mathcal{P}(R)$). /* See Figure 5-9 and Figure 5-10. */
$\mathcal{I} = \mathcal{P}(R)$.

while (true) do;
if (two or more registers in $\mathcal{I}$ have a common source) then do;
Let T denote the common source, and let i denote the distance.
Let $\mathcal{I}_{corr} \subset \mathcal{I}$ be the set of registers for which register T is a distance-i source.
/* Update the partition input set by replacing the correlated registers with their parents. */
$\mathcal{I} = (\mathcal{I} \setminus \mathcal{I}_{corr}) \cup$ *ParentsOfRegisterSet* ($\mathcal{I}_{corr}$) .
/* Update the expression for R's state to be in terms of the new partition input set. */
$\mathcal{E}$ = *ExpandExpressionTree* ($\mathcal{I}_{corr}$). /* See Figure 5-9 and Figure 5-10. */
end;

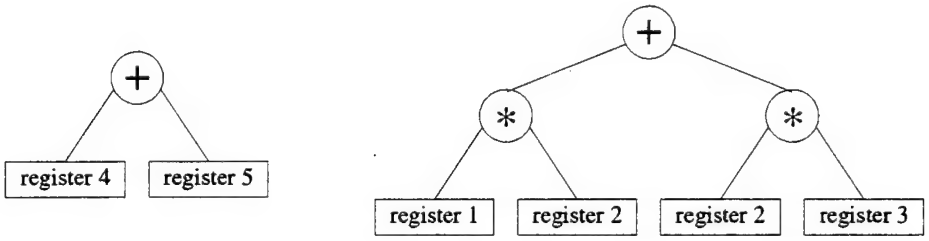
/* R is written in terms of a set of uncorrelated source registers. */
else return;
end;

Figure 5-8. Algorithm to transform and partition a circuit.

Algorithm *ExpandExpressionTree*
Input: An expression tree $\mathcal{E}$ and a set of registers $\mathcal{S}$.
Output: A modified expression tree (expanded around the registers in $\mathcal{S}$).

for each leaf register L in the tree do;
  if ($L \in \mathcal{S}$) then do;
    /* Expand the tree around this leaf. */
    Replace this leaf of the tree with a subtree expressing the state of L in terms of the states of the registers in $\mathcal{P}(L)$.
  end;
end;

Figure 5-9. Algorithm to expand an expression tree.



(a) The initial expression tree.

(b) expansion around registers 4 and 5.

Figure 5-10. Expansion of an expression for register 6.

the *ExpandExpressionTree* algorithm of Figure 5-9; these expansions are shown in part (b) of Figure 5-10. Our new input register set is $\mathcal{I} = \{R1, R2, R3\}$. At this point, we are done, since there is no correlation among the registers of $\mathcal{I}$.

5.2.4 A more complex transformation example

This subsection provides a detailed example of the transformation and partitioning algorithm by finding the proper partition for register 1 of the circuit of Figure 5-11. The circuit is shown completely transformed and partitioned in Figure 5-12. When

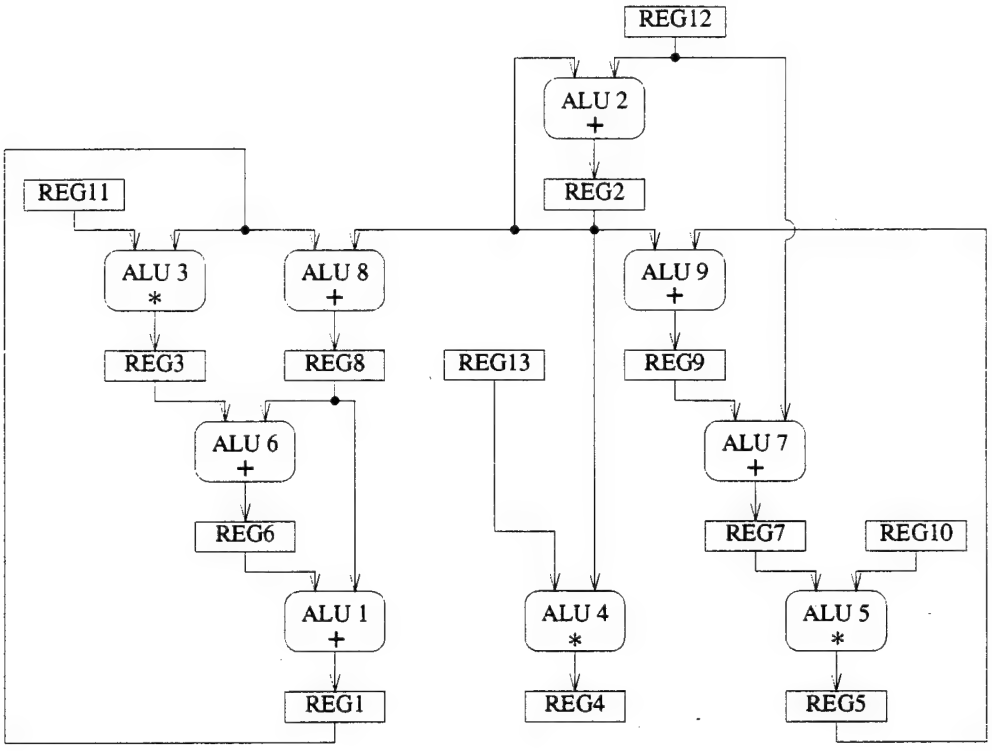


Figure 5-11. An example RTL circuit.

finding the partition for register 1, the algorithm begins by expressing the state of register 1 as a function of the states of its distance-0 sources, registers 6 and 8; this initial partition is shown in Figure 5-13(a). It then determines that registers 6 and 8 are correlated sources, since they have a distance-3 source in common; register 1 drives both register 6 and register 8 via paths (REG1 → REG8 → REG1 → REG3 → REG6) and (REG1 → REG3 → REG6 → REG1 → REG8), respectively. As a result, the expression for the state of register 1 is expanded around registers 6 and 8, to get the partition shown in Figure 5-13(b). Again, the algorithm looks for correlation among the input registers of the partition, and discovers that registers 8 and 2 are correlated;

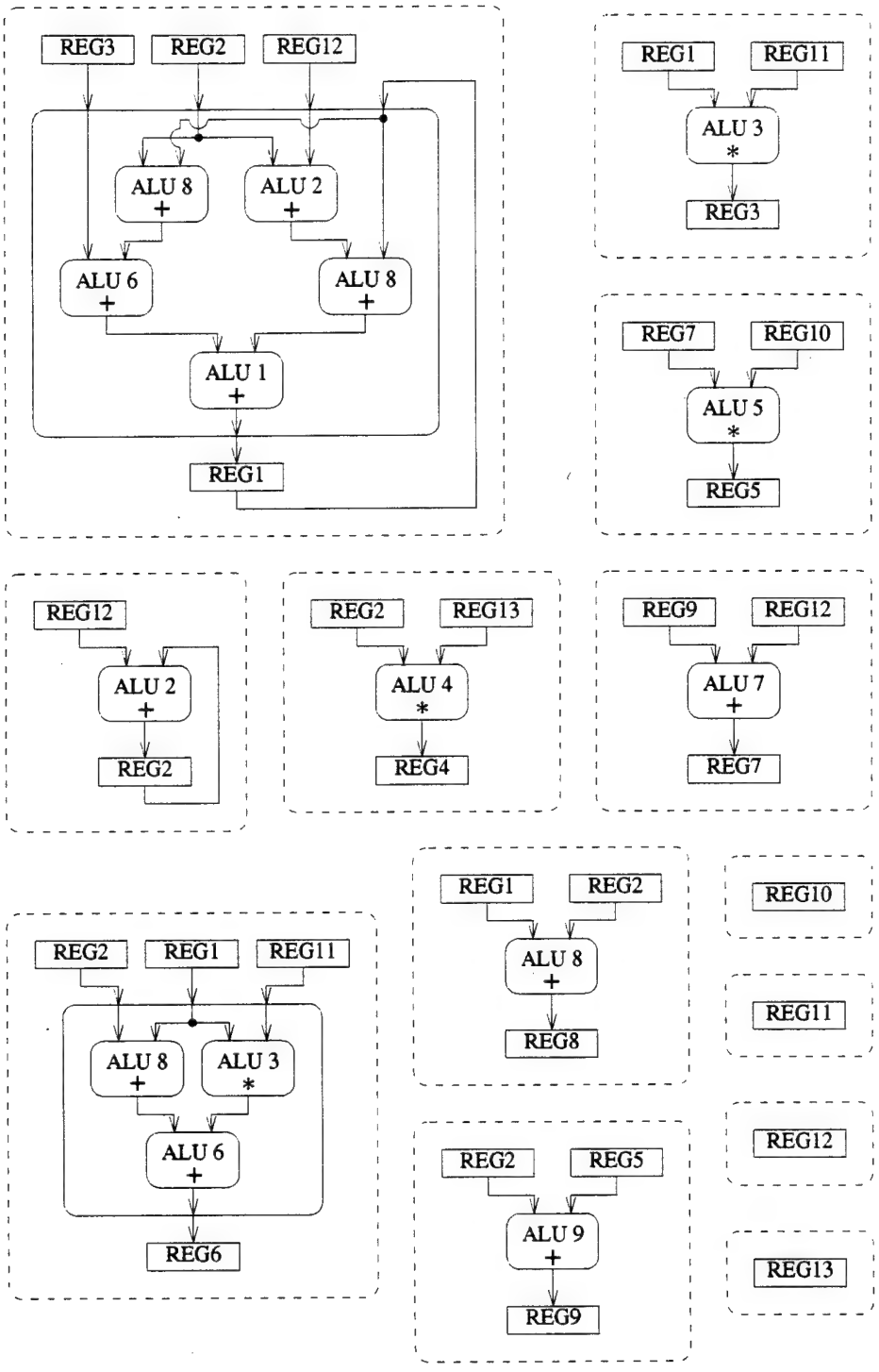


Figure 5-12. The circuit of Figure 5-11, transformed and partitioned.

register 2 is a distance-0 source of both registers. As a result, the expression is expanded around registers 8 and 2, giving the partition of Figure 5-13(c). After determining that no two input registers to this partition are correlated, the algorithm returns this partition as the final partition.

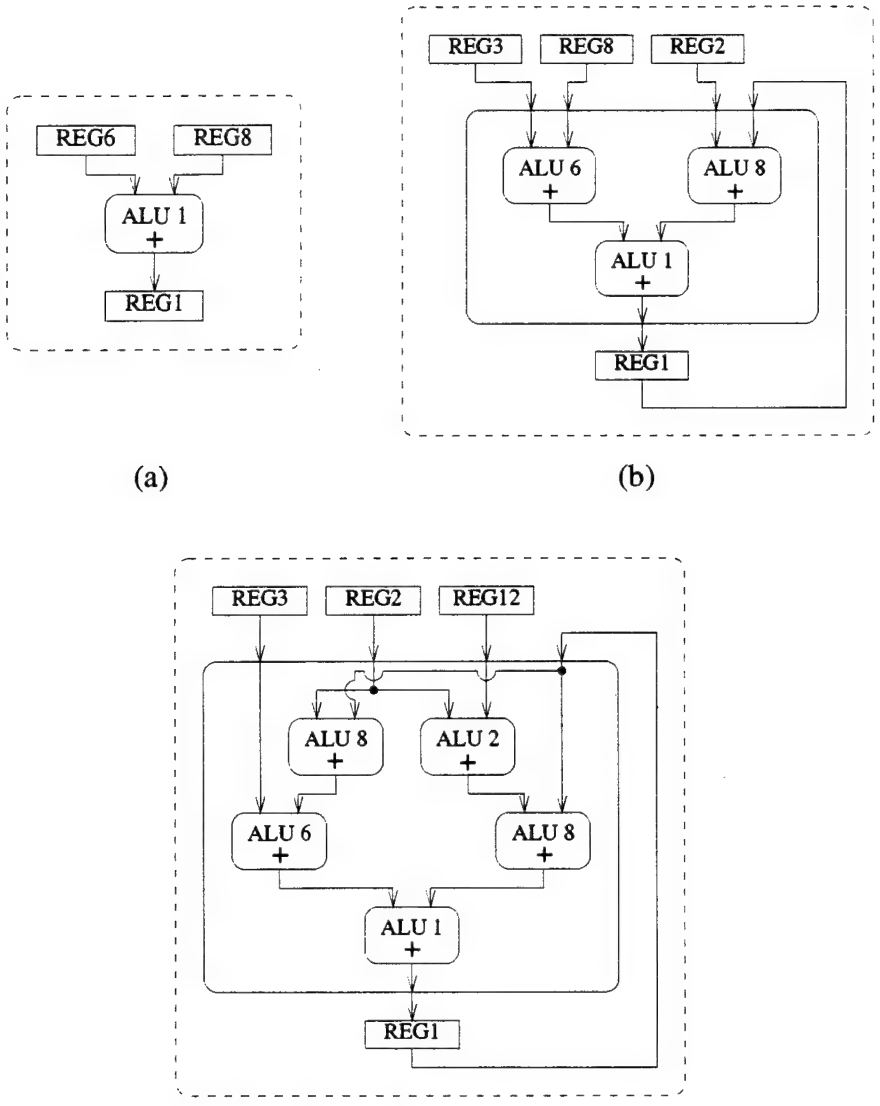


Figure 5-13. The steps in finding a partition for register REG1 of the circuit of Figure 5-11.

Note that the input registers to the partition are sources *of varying distance* from register 1. Register 3 is a distance-*1* source of register 1, while register 12 is a distance-*2* source. Thus, the partition expresses the state of register 1 at time *t* as a function of the states of register 3 at time *t*-2 and register 12 at time *t*-3. This does not present a problem for the analysis, since we deal with steady-state probability distributions, which do not change from time step to time step. Dummy registers can be inserted for other applications for which the timing must be preserved.

5.3 *Computational Complexity of the Analysis*

The overall execution time for analyzing a circuit is dominated by the time taken for the Markov analysis; the preprocessing transformation is very fast in comparison (less than a second of CPU time for the examples). We consider two components of the analysis separately: how the time for the overall analysis process described in Subsection 5.1.3 grows with the number of registers in the circuit, and how the time taken for the procedure analyzing a single register (Procedure *AnalyzeRegister*, given in Figure 5-2) grows with the bit width of the register.

The overall analysis process is iterative in nature. A single iteration analyzes each register in the circuit at most once, and so the time per iteration grows linearly with the number of registers. An exact complexity analysis of the overall process has not been done because determining the number of iterations needed is difficult, and would

require a numerical analysis technique. In general, circuits with indirect feedback take the most time, because they require more iterations. In our experience, most circuit require only a few iterations.

The time required to analyze a single register grows exponentially with the size of the register; this growth is inherent to the problem, since each possible register state must be examined. It should be noted that for real circuits, registers are of fixed size, and so it is important only that the analysis procedure be tractable for that size register. The overall execution time is reasonably fast for the four bit- and nine bit-wide register transfer level datapaths used as examples in Chapter Eight; our slowest example required less than four seconds of CPU time on a SUN SPARCstation *IPC* with 36 MB of memory. However, the procedure as it stands now is slow for larger registers, e.g. 16 or 32 bits. It is hoped that further research will find ways to speed up the analysis. Even as it stands now, the model provides insight into test quality, and thus is useful in validating faster heuristics; our research is exploring heuristics to compute the randomness and transparency metrics directly, without reliance on the underlying vectors and matrices.

5.4 The Markov Model at the Gate Level

Up to this point, this chapter has described a Markov model for the analysis of register transfer level circuits. We now shift focus to examine the special case when we have a gate level circuit instead. We start by grouping the individual gates of the gate level circuit into combinational logic segments, so that we can produce a "pseudo-RTL" view of the circuit. At this point, the procedure for calculating the 1-probability of one flip-flop in a gate level circuit is analogous to calculating the probability distribution of a register in a RTL circuit, shown in Figure 5-2. The only differences are that at the gate level, signals are one bit wide, so the probability distribution for a signal's state, which is represented as a vector at the register transfer level, can be represented with a single number, the *1-probability* for the signal. In addition, the state transition matrices used by the Markov model are now always two by two. These differences mean that the Markov model analysis done at the register transfer level can be replaced with simple scalar equations at the gate level. These equations provide analytical values for the 1-probability of each flip-flop in the circuit; entropies may be derived from the 1-probabilities using Equation 4-1 on page 47.

This section begins by describing how to create the "RTL" view of the gate level circuit. It then describes a computation method for the 1-probabilities of the signals throughout a circuit. Finally, it shows how this computation method is derived from the more general Markov model of Chapter Five.

5.4.1 Creating a "RTL" view of a gate level circuit

Before we can do a probabilistic analysis of a gate level circuit, we must identify the cone of logic that is driving each flip-flop and primary output of the circuit. This allows us to create a "RTL" view of the gate level circuit. Actually, "flip-flop transfer level" may be a more appropriate term, since the components of our "RTL" view are single flip-flops rather than registers, and combinational logic *cones*, or blocks with a single bit output rather than typical combinational logic blocks that have more than one bit of output. Note that if the output of a gate fans out, it may drive more than one of the flip-flops in the circuit. In this case, that gate will appear in more than one of the logic cones in the "RTL" view. An example of this process will be given in Chapter Seven, when we use Markov analysis to do BIST insertion in a gate level submodule of an industrial design.

As an aside, we mention here some recent work, presented in [PaBN94], that derives a true register transfer level circuit from a gate level circuit. The method begins in the same way we do, by grouping the gates into the logic cones that drive each flip-flop and primary output. It then attempts to look for logic cones that have similar functions so that it can group individual flip-flops into registers in a meaningful way. For example, if it notices that a number of the logic cones are bit slices of an adder, it will group those bit slices together, joining the flip-flops at the outputs of the cones into a register. Note that even in the general case, when there is no clear relationship between the logic cones, it is always possible to group flip-flops together to do a true register trans-

fer level analysis by allowing general "arithmetic logic units." However, this work is most meaningful if there is some underlying register transfer level structure inherent to the gate level circuit.

5.4.2 1-probability computation

This subsection presents a procedure for the computation of 1-probabilities of various signal lines in a sequential circuit. Note that if the circuit were exclusively combinational, we could find 1-probabilities for the lines of the network using the signal probability algorithm of Parker and McCluskey (see [PaMc75], [BaMS87]). This algorithm expresses the 1-probability at the output of a combinational logic block as a function of the 1-probabilities of the inputs of the combinational logic block. The key to the algorithm are input/output signal probability relationships for the basic gates, shown in Table 5-1. However, since we will be working with sequential circuits, we will require more than input-output relations for gates; we must also be concerned with the 1-probabilities of the values stored in the flip-flops, since the signal probabilities depend not only on the random inputs, but also on the state of the flip-flops of the CUT. The Markov model expresses the next state of a flip-flop of the circuit as a function of the current state. As we shall see in the next subsection, computing the flip-flop 1-probabilities necessitates classifying the flip-flops in terms of whether they feed back directly into themselves, and whether they are test registers included in the circular self-test path (for circuits using the circular BIST methodology). The Markov

model yields the set of formulas for the computation of flip-flop 1-probabilities shown in Table 5-2, where the notation used is defined in Table 5-3. The signal probabilities p_{CL}, p_{CL0}, and p_{CL1} are computed using the Parker-McCluskey algorithm.

We would like to underscore one fact about the formulas of Table 5-2 for circular BIST test flip-flops. Note that in both the feedback and no feedback cases, the 1-probability of a circular BIST test flip-flop is equal to $\frac{1}{2}$ as long as p_G, the 1-probability of the preceding flip-flop in the circular BIST chain, is $\frac{1}{2}$. In the case of circular BIST, this means that if the first flip-flop in the chain of test flip-flops is driven by a good quality test pattern generator, all of the flip-flops in the chain will generate highly pseudorandom test patterns, *regardless of the nature of the circuit under test.* For the circular self-test path technique, for which the ends of the chain of test flip-flops are connected to form a circle, if one flip-flop in the circular self-test path has good randomness properties, all of the flip-flops in the circular self-test path will have good randomness properties.

5.4.3 An example of 1-probability computation

This subsection presents a small example of the 1-probability computation. The example circuit of Figure 5-16(a), which builds a JK flip-flop out of a D flip-flop and some combinational logic, has one flip-flop that feeds back directly into itself.

gate type	input 1-probabilities	output 1-probability
k-input AND	$p_1, p_2, \ldots, p_k$	$\prod_{i=1}^{k} p_i$
k-input OR	$p_1, p_2, \ldots, p_k$	$1 - \prod_{i=1}^{k} (1 - p_i)$
k-input NAND	$p_1, p_2, \ldots, p_k$	$1 - \prod_{i=1}^{k} p_i$
k-input NOR	$p_1, p_2, \ldots, p_k$	$\prod_{i=1}^{k} (1 - p_i)$
2-input XOR	p_1, p_2	$p_1 + p_2 - 2 p_1 p_2$
NOT	p	$1 - p$

Table 5-1. Input/output signal probability relations (for combinational logic).

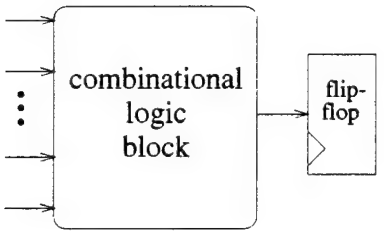
The first step is 1-probability computation for the state of the flip-flop. Since the circuit has direct feedback, two conditional 1-probabilities, p_{CL0} and p_{CL1}, corresponding to cases in which the feedback has value '0' and value '1', respectively, are computed. The appropriate circuit for the computation of p_{CL0} is shown in Figure 5-16(b); the annotated 1-probabilities of the signal lines come from a straightforward application of the gate relationship formulas of Table 5-1. For example, since the inverter on the K line has an input 1-probability of 0.5, its output probability is (1 - 0.5) or 0.5. The only exception is when the inputs to a gate are correlated due to reconvergent fanout; then,

flip-flop type	formula for 1-probability
no-feedback, normal	p_{CL}
no-feedback, circular BIST	$p_G - 2p_G p_{CL} + p_{CL}$
feedback, normal	$\frac{p_{CL0}}{1 + (p_{CL0} - p_{CL1})}$
feedback, circular BIST	$\frac{p_G - 2p_G p_{CL0} + p_{CL0}}{1 + (p_{CL0} - p_{CL1}) - 2p_G(p_{CL0} - p_{CL1})}$

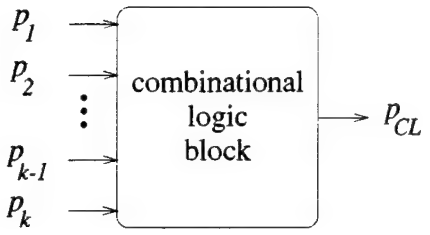
Table 5-2. Formulas for computation of flip-flop 1-probabilities (for sequential logic).

notation	meaning
p_{CL}	for a flip-flop F without direct feedback, the signal probability for the output of the combinational logic driving the flip-flop. See Figure 5-14(b).
p_{CL0}	for a flip-flop F with direct feedback, the conditional signal probability for the output of the combinational logic driving the flip-flop, given that the feedback line has value '0'. See Figure 5-15(b).
p_{CL1}	for a flip-flop F with direct feedback, the conditional signal probability for the output of the combinational logic driving the flip-flop, given that the feedback line has value '1'. See Figure 5-15(c).
p_G	for a test flip-flop F, the 1-probability of the flip-flop G that precedes F in the circular self-test path.

Table 5-3. Notation for 1-probability formulas.

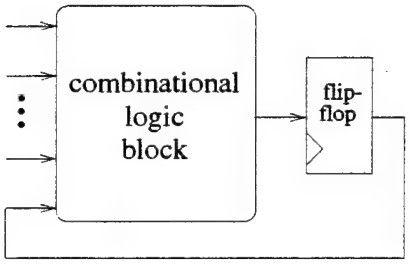


(a) a flip-flop that does not feed back into itself.

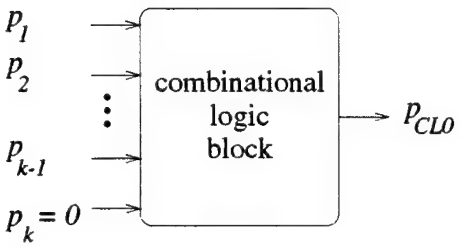


(b) the appropriate model for calculation of p_{CL}.

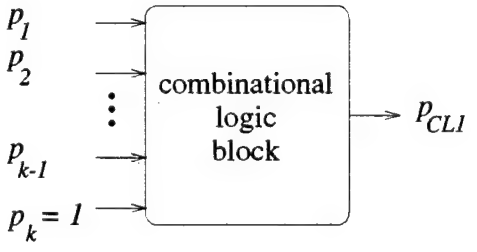
Figure 5-14. The no feedback case.



(a) a flip-flop that feeds back into itself.

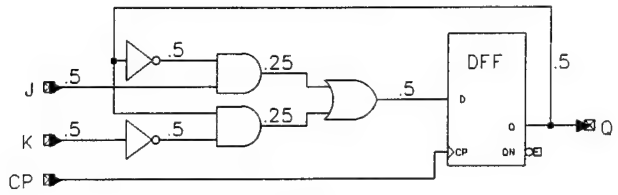


(b) the appropriate model for the calculation of p_{CL0}.

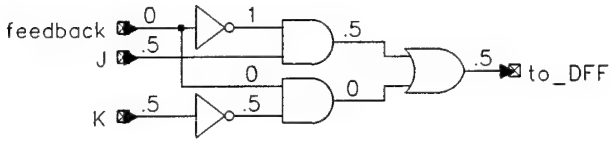


(c) the appropriate model for the calculation of p_{CL1}.

Figure 5-15. The feedback case.



(a) a circuit implementing a JK flip-flop.



(b) example calculation of p_{CL0}.

Figure 5-16. An example calculation.

a larger portion of the circuit, with uncorrelated inputs, must be considered. This is the case for the OR gate; to find the 1-probability at the output of the OR gate, we must write a truth table showing the output's value as a function of J, K, and the feedback line. Each line of the truth table corresponds to a different combination of values for J, K, and the feedback line; we can find the probability that a particular combination occurs from the individual 1-probabilities of J, K, and the feedback line. For example, the probability that combination (J, K, feedback) = ('1', '0', '1') occurs is

$$p_J(1-p_K)p_{\text{feedback}}.$$

The 1-probability at the output of the OR gate is found by summing the probability of occurrence of all those input combinations that result in a output of '1'.

From the figure, we see that p_{CL0}, the 1-probability at the output of the combinational logic segment, has value 0.5. A similar computation, this time setting the 1-probability of the feedback line to 1, gives a value of 0.5 for p_{CL1}. Finally, the 1-probability of the flip-flop is computed by using the appropriate formula from Table 5-2:

$$p_F = \frac{p_{CL0}}{1 + (p_{CL0} - p_{CL1})} = \frac{.5}{1 + (.5 - .5)} = .5.$$

Once p_F, the 1-probability for the flip-flop, has been computed, the (unconditional) 1-probabilities for all signal lines can be recomputed using the Parker-McCluskey algorithm, this time using the actual value for p_F instead of a constant 0 or 1. These final 1-probabilities are shown annotated to each signal line of Figure 5-16(a).

5.4.4 Iterating to analyze the entire circuit

As was the case with register transfer level circuits, an iterative procedure must be used to resolve indirect feedback. For exclusively feed-forward circuits, only one iteration is needed, but for circuits with circular BIST or indirect feedback loops, an iterative process analogous to that of Figure 5-5 is used to assign arbitrary initial 1-probabilities to the flip-flops and then repeatedly analyze flip-flops until the 1-probabilities have reached steady-state values.

5.4.5 A note about computational complexity

The original Parker-McCluskey algorithm for signal probability calculation has complexity that grows exponentially with the size of the circuit in the worst case. An extension by Savir et. al. [BaMS87] reduces complexity by cutting reconvergent fanout in the circuit, thereby turning the circuit into a tree. In doing so, exactness is sacrificed; the cutting algorithm produces upper and lower bounds, rather than exact signal probabilities. For the industrial subcircuit that we use as an example in Chapter Seven, we were able to apply the original, exact algorithm; however, for larger circuits, a technique like Savir's, which has complexity that grows linearly with the size of the circuit, may be more appropriate.

The next subsection provides analytical derivations for the 1-probability formulas of Table 5-2 in terms of a Markov model for the state of the flip-flops in the sequential circuit.

5.4.6 Derivation from the Markov model

This subsection presents a mathematical derivation for the formulas of Table 5-2 in terms of a Markov model. The Markov model describes the operation of a sequential circuit with circular BIST, using an "RTL" view of the circuit. The major difference that distinguishes the Markov model at the gate level from the Markov model at the

register transfer level is that at the gate level all state transition matrices are two by two and each signal's state probability distribution vector can be replaced with a single number, the 1-probability of the signal.

Step 1. Q for gate level circuits

The first step in analyzing a flip-flop is to compute a matrix Q describing the flip-flop's transitions from state to state during normal (BIST-disabled) operation. The form of the matrix Q depends on whether the flip-flop feeds directly back into itself. For a flip-flop F that does not feed back directly into itself (see Figure 5-14), we have simply that:

$$\begin{aligned} q_{ij} &= \Pr\{\text{next flip-flop state is } j\} \\ &= \Pr\{\text{the combinational logic driving } F \text{ has output value } j\} \end{aligned}$$

for $i \in \{0, 1\}$. This probability can be found by applying the Parker-McCluskey algorithm to the combinational logic driving F [BaMS87]; thus, we have:

$$Q = \begin{bmatrix} 1-p_{CL} & p_{CL} \\ 1-p_{CL} & p_{CL} \end{bmatrix} \qquad \text{(EQ 5-8)}$$

where p_{CL} is the 1-probability for the output of the combinational logic driving flip-flop F.

A flip-flop F driven directly by a primary input is a special case of the no-feedback class. Since during test flip-flop F is driven directly by a TPGR that generates random patterns with a uniform distribution, F has state transition matrix

$$Q = \begin{bmatrix} .5 & .5 \\ .5 & .5 \end{bmatrix}.$$

This can easily be modified to accommodate the use of *weighted random patterns* [BaMS87]; if, for example, a particular pseudorandom input is weighted so that it has value '1' three-quarters of the time, the proper state transition matrix is:

$$Q = \begin{bmatrix} .25 & .75 \\ .25 & .75 \end{bmatrix}.$$

Flip-flops that feed back directly into themselves must be handled differently. Consider a flip-flop F that feeds directly back into itself (see Figure 5-15). As in the previous case, the flip-flop F's new state comes from the output of the combinational logic driving F. The computation of

$$q_{ij} = \Pr\{\text{next flip-flop state is } j \mid \text{current flip-flop state is } i\}$$

requires two separate applications of the Parker-McCluskey algorithm, one in which the feedback has been assigned a constant value of '0' (yielding p_{CL0}), and one in which the feedback has been assigned a constant value of '1' (yielding p_{CL1}). We then have:

$$Q = \begin{bmatrix} 1-p_{CL0} & p_{CL0} \\ 1-p_{CL1} & p_{CL1} \end{bmatrix}. \qquad \text{(EQ 5-9)}$$

Step 2. C for gate level circuits

Step 2 of the analysis procedure of Figure 5-2 modifies Q, the flip-flop's state transition matrix for the BIST-disabled case, to compute C, the state transition matrix for the flip-flop when BIST is enabled. Note that if the flip-flop is not in the circular self-test path, in test mode it operates exactly as it does in normal mode, and therefore $C = Q$. For circular BIST test flip-flops, the elements of C are obtained by application of Equation 5-6:

$$C = \begin{bmatrix} q_{00}(1-p_G) + q_{01}p_G & q_{00}p_G + q_{01}(1-p_G) \\ q_{10}(1-p_G) + q_{11}p_G & q_{10}p_G + q_{11}(1-p_G) \end{bmatrix}, \qquad \text{(EQ 5-10)}$$

where p_G is the 1-probability of the preceding flip-flop G in the chain.

Step 3. Deriving the 1-probability

Step 3 of the analysis procedure of Figure 5-2 uses C, the flip-flop's state transition matrix when BIST is enabled, computed in Step 2, to compute the flip-flop's 1-probability in the steady state. All of the analysis done at the register transfer level applies here as well; however, since C is two by two, it is possible to write the 1-probability as a simple function of the elements of C, rather than in terms of a QR decomposition. Suppose that p_F denotes F's 1-probability in the steady state. Equation 5-7 becomes:

$$\begin{bmatrix} 1-p_F & p_F \end{bmatrix} = \begin{bmatrix} 1-p_F & p_F \end{bmatrix} \begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix}.$$

In solving the system, we find that

$$p_F = \frac{c_{01}}{1 - c_{11} + c_{01}} = \frac{1 - c_{00}}{1 - c_{00} + c_{10}} . \qquad \text{(EQ 5-11)}$$

Note that we have two equations in one unknown; it is always the case that C is rank deficient, so that both equations can be solved with the same value of p_F.

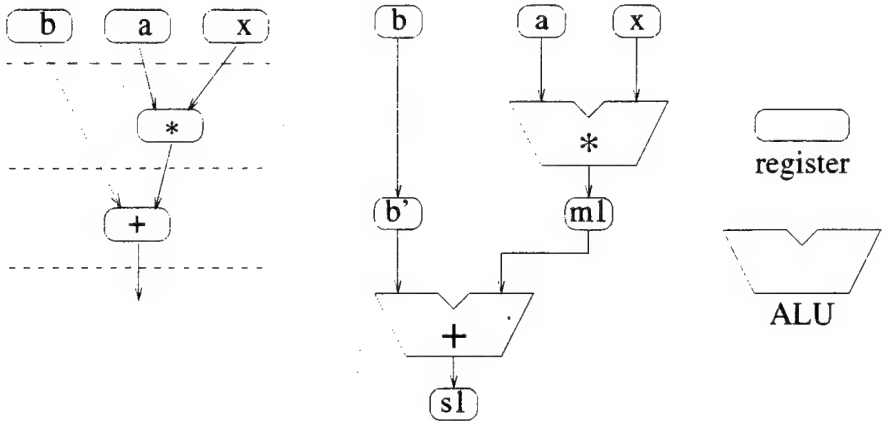
The formulas of Table 5-2 are found by simple substitution of the values for the elements of C into Equation 5-8 for the four separate classes of flip-flops. For example, consider a normal (non-test) flip-flop that does not feed back into itself. Since $C = Q$ in this case, we substitute the elements of Q, shown in Equation 5-8, into Equation 5-11:

$$p_F = \frac{c_{01}}{1 - c_{11} + c_{01}} = \frac{p_{CL}}{1 - p_{CL} + p_{CL}} = p_{CL} .$$

5.5 *The Markov Model at the Algorithmic Level*

This section describes how the same Markov model that was used in Section 5.1 to analyze circuits at the register transfer level (RTL) can be used to analyze algorithmic descriptions of circuits in the behavioral domain. Such descriptions consist of signals and operators, and are commonly depicted using *data flow graphs*; Figure 5-17(a) shows a data flow graph for the behavior $ax+b$.

We do our data flow graph analysis by mapping the data flow graph into an equivalent RTL circuit and analyzing the RTL circuit using the Markov chain model already described. We begin the mapping by choosing a schedule for the data flow graph; this



(a) data flow graph. (b) RTL structure.

Figure 5-17. One-to-one mapping between a scheduled data flow graph and the RTL structure used to analyze the graph.

requires assigning the operator nodes of the data flow graph to time steps. Our choice for the data flow graph of Figure 5-17(a) is designated on the figure by separating the time steps with broken lines. Note that there are many ways to schedule a given data flow graph; it does not matter which one we choose, because the results of the analysis will be the same for all of them. Figure 5-17(b) shows the RTL mapping for the example.The mapping replaces each operator node in the data flow graph with an arithmetic logic unit (ALU). Each edge of the graph, which represents an n-bit wide signal, is mapped to a number of n-bit wide registers in series. The number of registers is determined by counting the number of time step boundaries that the edge crosses. For example, the signal coming from primary input b crosses over two time step boundaries before it enters the adder, so it is mapped to two registers in the RTL circuit, labelled b and b'. The mapping is done this way so that correlation due to reconvergent fanout in the data flow graph is preserved in the register transfer level circuit. Note that

this mapping is solely for the purpose of analysis; we do not propose physically implementing the circuit this way. We then analyze the RTL circuit, and transfer the randomness, expected state coverage, and transparency metrics from the registers of the RTL circuit back to the edges of the data flow graph.

5.6 Summary

A Markov model for the analysis of conventional and circular BIST was presented. The model is applicable at a variety of levels of design abstraction; mathematical details were provided for the gate level, the register transfer level, and the algorithmic level. In addition, a transformation technique, used to process the circuit before applying the Markov model, was presented; this transformation technique strengthens the analysis by allowing accurate modeling of the effects of word-level correlation within the circuit. The model will be used to provide analytical measures for our testability metrics: randomness, expected state coverage, and transparency. Chapter Eight will include a number of examples for which the analytical values for the testability metrics from the Markov model are compared to actual results obtained from simulation, to the end of showing that the Markov model accurately captures what happens within a circuit.

The testability metrics computed using the Markov model will be used throughout the rest of this dissertation. In Chapter Six, randomness and expected state coverage will be used as tools to examine some structural difficulties inherent to the circular self-test path technique. In Chapters Seven, Eight, and Nine, the testability metrics will be used to guide BIST insertion at the gate level, register transfer level, and algorithmic level, respectively. The testability metrics provide a means of measuring the effect of a test insertion decision (concerning which signals should correspond to test registers or test flip-flops) on test quality.

Chapter 6

Structural Constraints on Circular Self-Test Paths

This chapter develops constraints on the structure of circular self-test paths in register transfer level (RTL) circuits with circular built-in self-test (BIST) features. The constraints arise from the desire to avoid *bit-level correlation*, which can have a devastating effect on test quality. Two causes of bit-level correlation are examined, with examples demonstrating the resulting degradation in test quality. The first cause of bit-level correlation, register adjacency, is a by-product of the ordering of the registers within the circular self-test path. The second cause of bit-level correlation stems from the shifting nature of the circular self-test path.

The work described in this chapter goes beyond previous work in a number of ways. First, the concept of register adjacency is generalized to arbitrary distances. Second, a newly discovered source of bit-level correlation in circuits using the circular self-test path technique is described. This correlation is a natural consequence of the shifting of bits in the circular self-test path; its presence greatly increases the probability of *system state cycling*, which can have a devastating effect on test quality. Guidelines for

avoiding this source of correlation by placing a new constraint on the structure of the circular self-test path are presented.

This chapter is organized as follows. Section 6.1 reviews the circular self-test path insertion process, which is the motivation for our work. Section 6.2 covers material preliminary to the main topic of the chapter; it describes how the system state transition graph can be used as a tool in understanding system state cycling. Section 6.3 describes a structural constraint on the circular self-test path arising from the desire to avoid bit-level correlation caused by register adjacency. Section 6.4 describes a constraint arising from a bit-level correlation that occurs because of the shifting nature of the circular self-test path. Section 6.5 is a summary.

6.1 *Circular Self-Test Path Insertion*

The process of inserting circular self-test path features into a register transfer level circuit requires two main steps. The first is the *selection* of registers for the circular self-test path; this can be accomplished in a number of ways, ranging from selecting a *maximal path* of all registers, to selecting a *minimal path* of only the primary input and primary output registers. Selection of registers can also be done using heuristics based on circuit structure and testability measures, similar to what [POLB88] and [LiZB93] have done at the gate level. In Chapter Eight, we will do BIST insertion at the register

transfer level based on testability metrics, computed using the Markov model described in Chapter Five.

The second step of circular self-test path insertion is the *formation* of the circular self-test path from the selected registers. A circular self-test path consists of an *ordered* and *oriented* set of circuit registers. An algorithm to link the selected registers into a circular self-test path must choose both the ordering for the registers in the path, and the orientation or *shift direction*, right or left, for each register in the path.

Analysis of the quality of the circular self-test path technique as a testing technique so far has assumed that there is no statistical correlation between the streams of bits applied to the two inputs of any of the exclusive OR (XOR) gates in the circular self-test path [PiKK92]. Figure 6-1 shows a single flip-flop from the circular self-test path. The assumption is that bits x and y are not correlated. Unfortunately, for some circular self-test path structures, this assumption does not hold; *bit-level correlation* occurs, and test effectiveness can be severely affected. It is important to identify the structures that cause bit-level correlation so that they can be avoided during formation of the circular self-test path. This chapter examines two causes of bit-level correlation, and the limitations that must be placed on the structure of the circular self-test path to avoid the correlation. Preliminary to the discussion of these constraints, the system state transition graph, a tool for understanding the problem of system state cycling, is described.

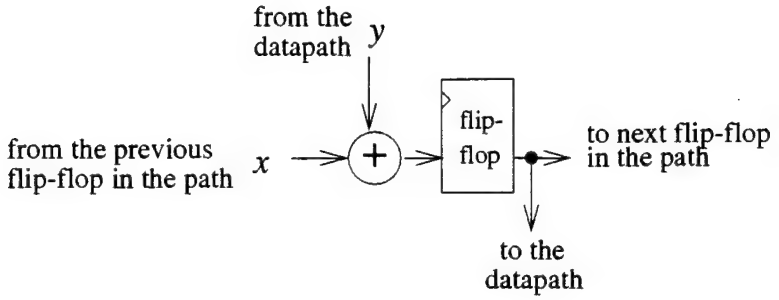


Figure 6-1. A single flip-flop of the circular self-test path.

6.2 *System State Cycling*

One problem that has plagued the circular self-test path technique is that of *system state cycling*, or *limit cycling*. The state of a system is determined by the contents of the system's memory elements. Following the definitions of Brynestad, Aas, and Vallestad [BrAS90], let the system *state transition graph* (STG) of a circuit be a directed graph, where the nodes represent system states and the edges represent transitions between system states. An example shape of an STG is shown in Figure 6-2. When in test mode, the system as a whole is deterministic; since no data enter the circuit via the primary inputs, given the current state, there is one unique next state. This means that there is exactly one edge leaving each node. Note that there may be more than one edge entering a node. In general, the STG is partitioned into one or more *cycles* or *rings*, which may have *tails*, or paths of states that are not part of the cycle itself, but eventually lead to the cycle. The first node of each tail has special significance; since such a node has no edges leading into it, it corresponds to a state that can not be gener-

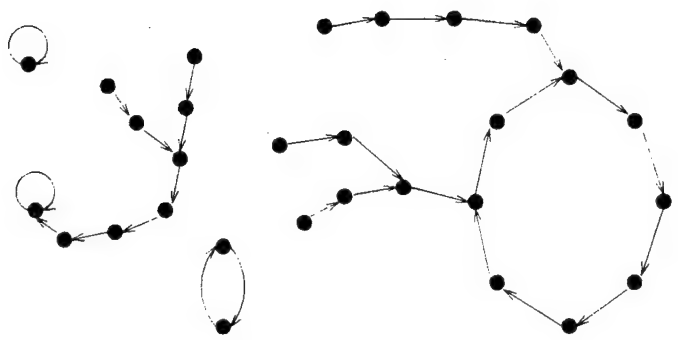


Figure 6-2. Example shapes for the STG.

ated naturally during a test session. We call such a state an *unreachable* state; in [BrAS90], these states are called *leaves*.

Each test session begins by putting the system in an initial state; this state is called the *system seed*. This corresponds to picking one node of the STG. Note that it is possible to pick an unreachable state as the system seed, since the system is not in test mode when we initialize it. Next, the system is clocked; each clock corresponds to moving along one edge of the STG to another system state. If the system is clocked enough times, it will end up in a state that it has already visited. From that point on, all system states generated will be repeats of previously generated states; on the STG, a path around and around a cycle is taken, and we say that the system has entered a cycle. Note that the test patterns applied to a given combinational logic block also start cycling at this point, since the bits in a test pattern are a subset of the bits in the system state. As a result, no new test patterns for the combinational logic blocks are gener-

ated. Thus, if a system enters a cycle early in a test session, test quality will be low regardless of the length of the session.

All deterministic systems, including circuits utilizing conventional BIST techniques, are subject to system state cycling; however, *circular self-test path circuits are especially likely to enter a system state cycle early in a test session.* The shape of the STG indicates this susceptibility; ideally, we would like to choose our test session so that we travel along a long path in the STG before we enter a cycle. If the STG is composed of a large number of small cycles, this may not be possible. Similarly, if the STG has a large number of tails, we may not be able to avoid cycling, since we can choose to travel along at most one of the tails. By looking at the relationship between certain circular self-test path structures and the resulting shape of the STG, we gain insight into why those structures can result in poor test quality.

6.3 Register Adjacency

One cause of bit-level correlation is *unit-distance register adjacency* [Stro88] [POLB88]. This phenomenon, which is illustrated in Figure 6-3, occurs when flip-flop f_j comes directly after flip-flop f_i in the circular self-test path, and is also fed directly by flip-flop f_i through the combinational logic of the datapath. From the figure, we see that unit-distance register adjacency in an RTL structure can be thought of as bit-level reconvergent fanout; the output of flip-flop f_i fans out, and the two copies of the bit

that was stored in f_i meet again, and are recombined, before being stored in f_j at the next clock. One copy of the bit takes a path through the combinational logic of the RTL structure; the other copy travels along the circular self-test path.

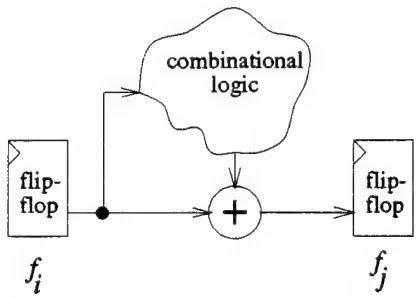


Figure 6-3. The concept of register adjacency.

Figure 6-4 shows a simple kind of register adjacency. Suppose that the most significant bit of register B has value x. On the next clock, this bit fans out to both inputs of the XOR feeding the most significant bit of register A; this correlation of inputs means that the most significant bit of A will have value $x \oplus x = 0$. This phenomenon occurs at each time step of a test session, holding the most significant bit of A constant at value 0, and having a devastating effect on the randomness of register A's state: fully half of the states can never be realized. This is reflected in the best achievable state coverage for register A, which is 0.50. In terms of the STG, we see that one half of the system states (i.e., all states for which the most significant bit of A is a 1) are unreachable, and are therefore at the beginning of a tail. For a circuit of four-bit registers, assuming that register B's states are perfectly pseudo-random (the best possible case),

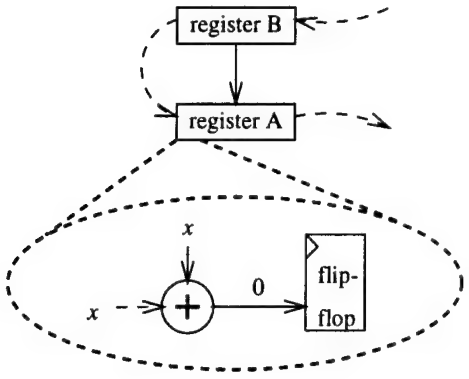


Figure 6-4. A portion of an RTL structure showing unit-distance register adjacency.

	register *B*	register *A*
randomness	1.00	0.75
best achievable state coverage	1.00	0.50

Table 6-1. Metrics for the registers of Figure 6-4.

register *A*'s randomness is reduced to 0.75, reflecting the fact that while three of the bits of *A* are pseudo-random, one bit is stuck at a constant value.

Figure 6-5 is another example of register adjacency. As in the last example, the adjacency of registers *B* and *A* causes a correlation between the inputs of the XOR gate that drives the most significant bit of *A*. To see why, let the most significant bits of registers *B* and *D* have values *y* and *x*, respectively. On the next clock, the XOR gate will have inputs *x* and *x*&*y*, for a result of $x\&\bar{y}$. At the same time, the *y* value will shift into the least significant bit of *B*. This means that there is a correlation between the most significant bit of register *A* and the least significant bit of register *B*; namely, if the least significant bit of *B* has value 1, the most significant bit of *A* must have value 0.

This dependency between bits of the system state holds throughout a test session, thereby limiting the system states that can be realized; any system state with 1's in both the least significant bit of *B* and the most significant bit of *A* is unreachable. A more comprehensive example, in Table 6-4 of Section 6.4, shows that the presence of so many tails in the STG gives circuits with this structure a tendency to cycle. The dependency also degrades the randomness of register *A*; if registers *B* and *D* are perfectly random, the most significant bit of register *A* will have value 1 only one-quarter of the time. As a result, when registers *B* and *D* have perfectly random states, register *A*'s randomness is decreased to a value of 0.94. Although it may still be possible to generate all 16 patterns in register *A*, doing so will require a longer test session. To illustrate this, consider the number of clocks that we must apply to the circuit to achieve an expected state coverage of 99% for register *A*, given that system state cycling does not prevent us from reaching this level of state coverage entirely. As the circuit is now, we must apply 146 clocks; if register *A*'s randomness were 1.0, in contrast, we would have to apply only 72 clocks.

Furthermore, if registers *A* and *B* serve as inputs to some other ALU in the circuit, the interdependency between the states of the two registers will make it impossible to exhaustively test that ALU. Specifically, any test pattern that has a 1 in the least significant bit of *B* and a 1 in the most significant bit of *A* can not be generated.

Unit-distance register adjacency is a consequence of the ordering of registers in the circular self-test path, and can easily be identified. We first construct a directed graph

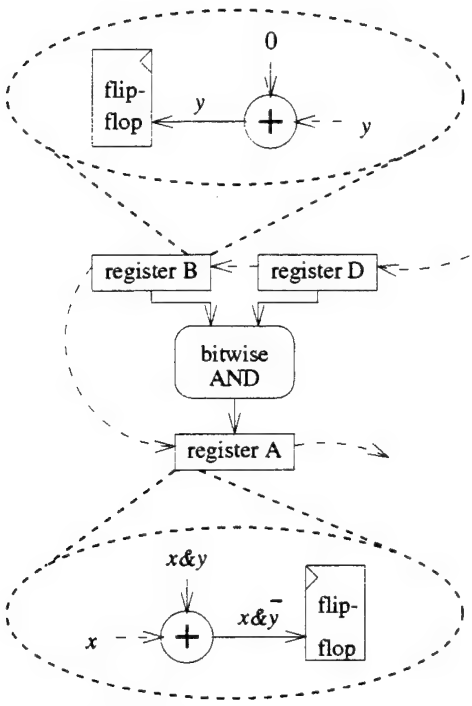


Figure 6-5. A second example of distance-1 register adjacency.

	register *B*	register *D*	register *A*
randomness	1.00	1.00	0.94
best achievable state coverage	1.00	1.00	1.00

Table 6-2. Metrics for the registers of Figure 6-5.

indicating the data flow from memory element to memory element; in this graph, the nodes correspond to the flip-flops of the circuit, and there is an edge from node f_i to node f_j if and only if flip-flop f_i feeds directly into flip-flop f_j through combinational logic. Next, we consider the flip-flops in the circular self-test path, two by two, in order. If flip-flop f_j directly follows flip-flop f_i in the circular self-test path and there is an edge from f_i to f_j in the graph, we have exactly the situation shown in Figure 6-3, and there is register adjacency in our circuit. Register adjacency can usually be

avoided by careful ordering of registers in the circular self-test path, using a technique like that described in [Stro88]. In the event that no ordering of registers is adjacency-free, an extra flip-flop can be inserted into the path to break up the adjacency. While this flip-flop is transparent in the normal mode, in test mode it serves to delay the values traveling around the circular self-test path by one clock [PiKK92]. An alternative approach for avoiding the bit-level correlation caused by unit-distance register adjacency involves modifying the design of the test register [AvMc93]. Another approach tests whether a particular instance of register adjacency has a significant effect on test quality before restructuring the path to remove it [Gage93].

The two previous examples are of *unit-distance* register adjacency: the bit value that fans out reconverges after *one* clock. However, register adjacency can occur at arbitrary distances. Distance-*d* register adjacency occurs when the bit value that fans out reconverges after *d* clocks. Figure 6-6shows a circuit with distance-2 adjacency. For this example, registers are two bits wide. Suppose that the least significant bit of register *B* has value *x*. In two time steps, the *x* value will fan out and travel to both inputs of the XOR that drives the least significant bit of register *A*; one copy of *x* travels along the circular self-test path, and the other travels through the datapath (via register *D*). As a result, the least significant bit of register *A* will have value $x \oplus x = 0$. This happens at each subsequent clock, after an initial set-up time of one clock. That is, this correlation does not occur the very first time the circuit is clocked, but it does occur at each clock after that. Once again, the randomness of register *A* is degraded.

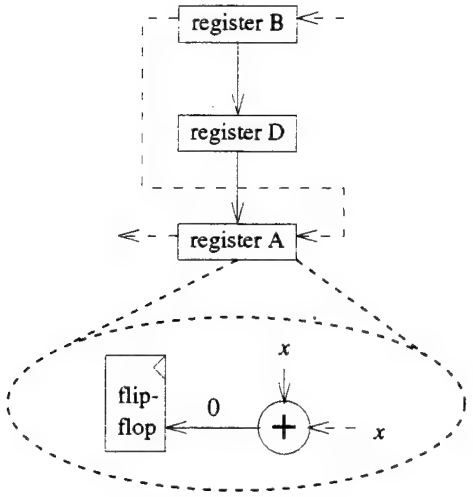


Figure 6-6. An example of distance-2 register adjacency.

	register *B*	register *D*	register *A*
randomness	1.00	1.00	0.50
best achievable state coverage	1.00	1.00	0.50

Table 6-3. Metrics for the registers of Figure 6-6.

An algorithm for identifying distance-*d* register adjacency must first extend the directed graph representing data flow from memory element to memory element by annotating the edges with distances. All of the original edges have distance 1. If in the original graph there is a path from node *i* to node *j* traversing *d* edges, an edge from *i* to *j* with distance *d* is added to the extended graph. Distance-*d* register adjacency exists between flip-flops f_j and f_i if f_j comes *d* flip-flops after f_i in the circular self-test path and there is a distance-*d* edge from *i* to *j* in the graph.

6.4 A Shift-Related Cause of Correlation

Register adjacency is not the only phenomenon that causes bit-level correlation; when an arithmetic logic unit in the RTL structure performs a bit-wise operation, correlation can arise as a natural consequence of the shifting of the registers in the circular self-test path. Consider the circuit fragment of Figure 6-7. Suppose that the least significant

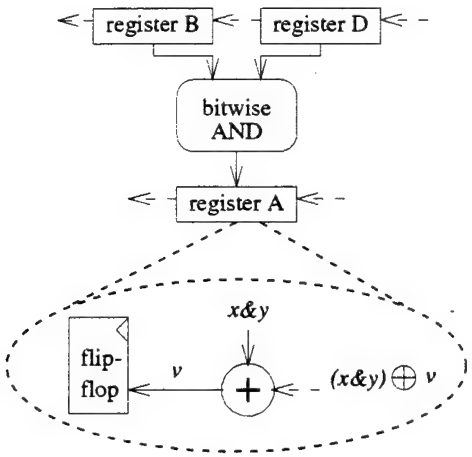


Figure 6-7. A portion of an RTL structure producing bit-level correlation due to shifting.

bits of registers *B* and *D* have values *x* and *y*, respectively. Since the primary inputs of the circuit are held constant at zero during a test session, registers *B* and *D* operate as ordinary shift registers. As a result, the *x* and *y* values travel to the left in lockstep as the circuit is clocked. This means that the product *x&y* is formed at successive outputs of the ALU. Correlation occurs when the product *x&y* from one clock meets with the product from the previous clock and cancels out. As shown in Figure 6-7, this happens in the second least significant bit of the output register *A*. The input to the XOR gate

from the ALU is simply the product $x\&y$; the input from the circular self-test path is the same product, formed one clock cycle earlier, XORed with a value from the circular self-test path, $(x\&y) \oplus v$. The product cancels out, leaving a result of v. A similar correlation occurs in every second bit of register A.

The difficulty with this type of correlation is that register A no longer acts like a multiple input shift register; the cancellation of terms causes A to act like an ordinary shift register from the point of view of the circular self-test path. In other words, the values shifted out of A onto the circular self-test path are simply a time-delayed version of the bits shifted into A. This may affect the randomness directly, and, more importantly, it makes the circuit extremely susceptible to system state cycling. Note that when the circuit is in test mode, the primary input registers all act as simple shift registers, since no inputs are presented to the circuit. The more registers that perform simple shifts instead of using the responses of the circuit to find the next state, the more likely the system is to cycle.

The shifting type of correlation described in this section occurs not only for bit-wise operations like AND and OR, but also for operations with a bit-wise component. For example, this same phenomenon occurs for the ADD function, which is essentially a bit-wise XOR with some carry information included. While the cancellation of terms for an ADD circuit is not as catastrophic as it is for our AND example, it is still enough to significantly degrade test quality.

Unlike the register adjacency case, this cause of bit-level correlation cannot be avoided by re-ordering the registers in the circular self-test path; for our example, it does not matter where in the path registers *B*, *D* and *A* lie. Instead, we must pay attention to the *shift direction* of the registers. If we shift input registers *B* and *D* in opposite directions, the *x* and *y* values no longer travel to the left in lockstep, meeting at an AND gate at each time step; instead, they meet only once. Similarly, we could choose to shift the output register in a direction opposite to that of the input registers; for this configuration, values *x* and *y* still travel in lockstep, but their product from one time step does not cancel out their product from one time step before. Therefore, when constructing a circular self-test path, it is important to take care that not all register ports of a bitwise operation are shifted in the same direction.

Table 6-4 shows the relationship between the choice of shift directions and test quality for four-bit AND and ADD examples. For each example, three circular self-test path configurations are used. The best achievable state coverage for the ALU and the longest achievable time before the system starts cycling are shown; these are found via simulation, using the initial system state or *seed* that gives the most favorable results. In the first configuration, all three registers are shifted in the same direction; this circuit suffers from the shifting correlation described above. Note that the AND example enters a system state cycle after only twelve clocks. The second configuration avoids the shifting problem, but introduces register adjacency. Here, we see that a large number of unreachable states, and therefore a large number of tails in the STG, result in

circuits that cycle badly; the AND example enters a system state cycle after 36 clocks, and the ADD example after 185 clocks. The third configuration shows that the shift directions can in fact be chosen to avoid both sources of bit-level correlation. The same kind of shifting correlation that occurs in the first configuration will also occur when this small example is embedded in a larger circuit.

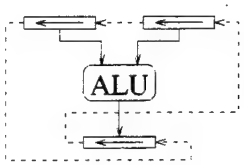
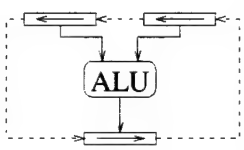
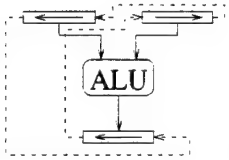
	 a bad choice; has bit-level correlation	 a bad choice; has unit distance register adjacency	 a good choice
best achievable state coverage for AND in one test session	$\frac{12}{256} = 0.047$	$\frac{36}{256} = 0.141$	$\frac{255}{256} = 0.996$
longest time to system cycle	12	36	2113
best achievable state coverage for ADD in one test session	$\frac{218}{256} = 0.852$	$\frac{139}{256} = 0.543$	$\frac{256}{256} = 1.000$
longest time to system cycle	1225	185	1739

Table 6-4. Shift direction and its impact on test quality.

Figure 6-8 shows shifting correlation within a larger circuit. For this example, the registers are four bits wide. As in the smaller example, shifting correlation causes the output register to act as a shift register from the point of view of the circular self-test path; bit-level correlation in the least significant bit and the third least significant bit of the

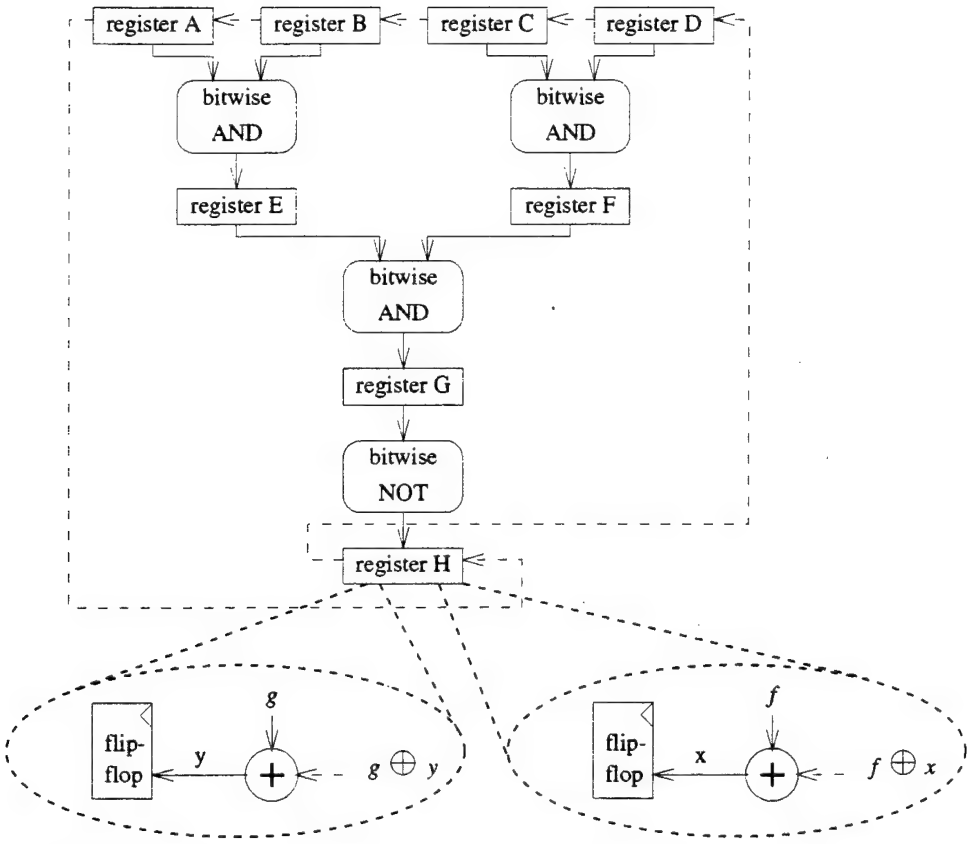


Figure 6-8. A second example of bit-level correlation due to shifting.

register ensure that the values shifted out of *H* onto the circular self-test path are simply a time-delayed version of the bits shifted into *H*. Note that for this circuit, then, all registers in the path are essentially shift registers. There are 20 flip-flops in the path; as a result, the state of the path will cycle in 20 clocks. This makes it extremely likely that the entire system will also cycle in 20 clocks, making this circuit impossible to test adequately.

Identifying shifting correlation in a circuit using the circular self-test path technique involves looking for sections of the circuit that perform bitwise, or largely bitwise,

operations. When such a section exists, shifting correlation will occur if all input and output registers of the section are shifted in the same direction. Therefore, care should be taken to choose a set of register orientations such that at least one of the input and output registers of the section is shifted in a direction opposite to that of the other registers.

6.5 Summary

Two sources of bit-level correlation in RTL circuits using the circular self-test path technique have been described, along with examples demonstrating the degradation in test quality resulting from the presence of the correlation. In avoiding these two sources of correlation, constraints are placed on the structure of the circular self-test path. The first source of correlation, register adjacency, can be avoided through careful choice of the ordering of the registers within the path. The second source of correlation, which arises from the shifting nature of the circular self-test path, can be avoided through careful choice of a shift direction, right or left, for each register in the path. A clear understanding of these sources of correlation, and the resulting effect in terms of both test patterns that can not be generated and the possibility of system state cycling, is the first step in automating the circular self-test path formation process.

Although the work of this chapter was done with the circular self-test path methodology in mind, much of it is also applicable to the circular BIST methodology. As was

reviewed in Chapter Two, the two methodologies are similar; both create a chain out of the same kind of test register. Register adjacency can occur in circular BIST circuits, and should be avoided using the techniques outlined in this chapter. The shifting correlation described in this chapter can also occur in circular BIST circuits. However, in the case of circular BIST, system state cycling does not become overwhelmingly likely when shifting correlation is present. The reason for this difference is that the circular BIST technique uses a test pattern generation register (TPGR) to drive the input end of the chain, and a multiple input shift register (MISR) to watch the output end of the chain, rather than simply recirculating the system bits the way the circular self-test path methodology does.

Chapter 7

BIST Insertion at the Logic Level

In industry today, design and test are often considered separate problems; design of a circuit is done without regard to testability, and BIST features are added to the design after the fact. The test quality of circuits designed this way can be quite poor; often, the underlying structure of the circuit makes it difficult to test using pseudorandom test patterns. When faced with such a *random pattern resistant* design, the circuit designer has two options. The first, to re-synthesize the circuit from scratch, this time keeping test in mind, is often undesirable. In some cases, well established circuits, known to work well, have been described informally, e.g., with block diagrams. Re-synthesis would require describing the function of such a circuit formally in the behavioral domain; however, the behavior may not be immediately known, or it may be difficult to capture the exact behavior in a formal description language such as VHDL. For these reasons, re-synthesis may involve a degree of risk that designers are unwilling to incur.

The second option, which is the only alternative when re-synthesis of a design can not be done, is to try to enhance the testability of the existing design. This second option is

the one explored in this chapter. The goal of this work is to present a *systematic approach* for testability enhancement of existing logic level designs utilizing pseudo-random BIST. The approach is to remove the resistance of the circuit to random patterns in test mode while changing as little of the design as possible. A testability metric, *entropy*, is used to pinpoint problem areas of the design; entropy is calculated using probabilistic analysis based on the Markov model presented in Chapter Five. Signals with low entropy can make testing difficult by affecting both controllability and observability of internal parts of the circuit.

This chapter next describes a methodology for built-in self-test (BIST) insertion in logic level circuits. The methodology is demonstrated on a submodule of an industrial logic level design. Fault simulation is used to show that the resulting modified circuit designs are indeed more easily tested than the original.

7.1 Testability Enhancement Technique

This section describes our testability enhancement procedure. We assume the test paradigm of Figure 7-1, i.e., we assume that in test mode the overall circuit is configured

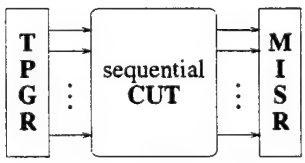


Figure 7-1. The basic BIST scheme used.

so that the inputs of a sequential module are driven by a test pattern generation register (TPGR), and the outputs of the module drive a multiple input shift register (MISR). In this way, we can focus on the sequential circuit under test (CUT), assuming that is driven by random patterns of known distribution. Essentially, we will be looking within a circuit for lines with 1-probabilities that are very close to either 0 or 1. Alternatively, we may look for lines of low *entropy*, as defined in Equation 4-1 on page 47. The 1-probabilities and entropies for various points in the circuit are calculated using the Markov model described in Chapter Five.

We will take two different approaches in dealing with problem areas in a design. The first approach involves the use of the circular BIST methodology within the sequential CUT. For this approach, the overall BIST scheme of Figure 7-1 is preserved, but some of the flip-flops internal to the CUT are used to form a circular BIST chain. The second approach involves the use of multiplexers to insert test points that alter the functioning of the circuit during test; a multiplexer, controlled by a signal that indicates whether the circuit is in normal or test mode, may serve, for example, to deliver high quality test patterns to a portion of the circuit that is not ordinarily highly controllable.

Our testability enhancement methodology, shown in Figure 7-2, is a two step technique. The first step involves the testability enhancement for flip-flops that have low entropy. Once the flip-flop entropies have been calculated, testability problems may become apparent; since the flip-flops drive the combinational logic, low entropy of the flip-flops translates directly into the application of poor quality tests to the gates of the

Algorithm *TestabilityEnhancement*

Input: A sequential circuit design.

Output: The circuit design, modified to enhance testability.

Step 1. Use the *Markov model* to calculate flip-flop entropies.

For those flip-flops with low entropy, do testability insertion to enhance the entropy.

If fault coverage of resulting design is satisfactory, stop here.

Step 2. Within the combinational logic segments, use the *Parker-McCluskey algorithm* to find entropies of internal nodes.

Select the line nearest the primary inputs that has low entropy. Enhance the entropy of that line by doing testability insertion. Recompute the internal node entropies.

If some internal lines still have low entropy, repeat the selection and enhancement part of this step.

Figure 7-2. Overview of testability enhancement technique.

circuit, and thus into low fault coverage for the combinational logic. Therefore, if any flip-flop in the CUT has low entropy, the situation should be remedied. For circuits using circular BIST, this may be accomplished by replacing the low entropy flip-flop with a test flip-flop that in test mode is part of the circular BIST chain. For circuits using multiplexer-based test point insertion, flip-flop entropy can often be raised by breaking loops. When Step 1 is complete, all the combinational logic of the sequential CUT is driven with high quality test patterns when in test mode.

If test quality of the circuit is still poor, it is necessary to take a closer look at the combinational logic segments, and to alter the structure of the combinational logic to

remove the random pattern resistance [BaMS87] that can make it difficult to test the combinational logic using random patterns. This is done in the second step of the testability enhancement methodology, which involves looking at the entropies of nodes internal to the combinational logic segments. We apply the Parker-McCluskey algorithm [PaMc75] to the combinational logic segments to find signal lines of low entropy. Again, there are a number of ways to improve the entropy of individual signal lines; we may choose to insert an additional circular BIST flip-flop that is transparent in normal mode, or to use a multiplexer to insert a test point.

7.2 Example Application

This section describes the circuit that will be used as an example to provide the details of the testability enhancement technique. The circuit, shown in Figure 7-3, is a submodule of an existing design provided by Rockwell International. The overall chip was designed by Rockwell engineers as a large number of submodules, each of which is a sequential circuit. The submodules are quite small, with typically about ten internal flip-flops. The submodules are interconnected via intervening registers.

As a first step, conventional BIST was added to the overall chip in the following simple way: the registers of the chip were replaced with special test registers that can act as test pattern generators (TPGRs) and multiple input shift registers (MISRs). Thus, we test each submodule separately, using the paradigm of Figure 7-1. Our goal is to

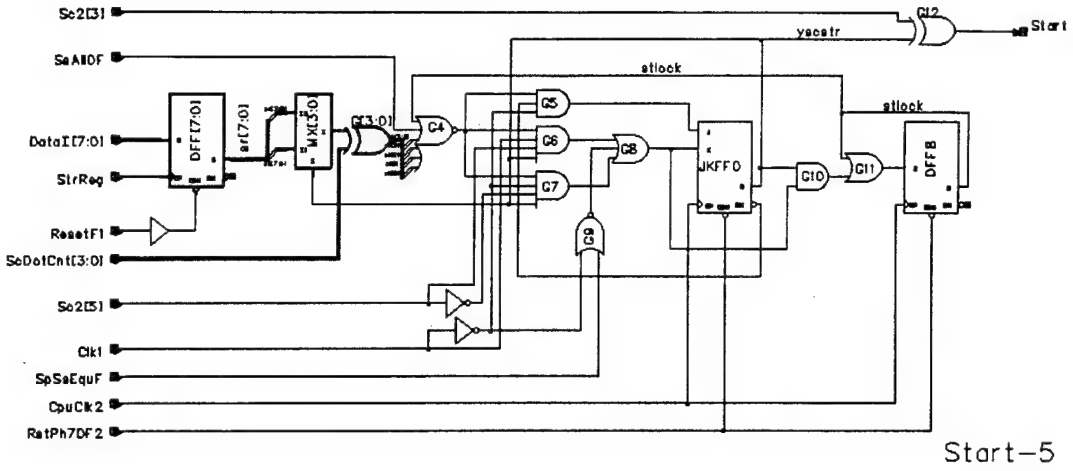


Figure 7-3. A submodule of an industrial design.

modify the submodules, if necessary, to make them random pattern testable. We demonstrate this by applying the testability enhancement technique of Figure 7-2 to the example submodule.

We consider two distinct sets of experiments. The first uses circular BIST within the sequential circuit, and the second employs multiplexer-based test point insertion. For both sets of experiments, we compare fault coverage, area, and critical delay for the original and enhanced variations of the circuit. Fault coverage data are obtained using AT&T's GENTEST fault simulator [Davi94]; area and performance figures come from the COMPASS Design Automation suite of tools [CODA92]. Areas are found by performing layout synthesis using standard cells. Performance is measured in terms of the critical delay for the circuit, and comes from a timing verifier in the COMPASS suite, using circuit models that include interconnect delay. Area and critical delay fig-

ures are for the sequential CUT only, and do not include the TPGR and MISR. Overhead for a circuit is calculated with the following formula:

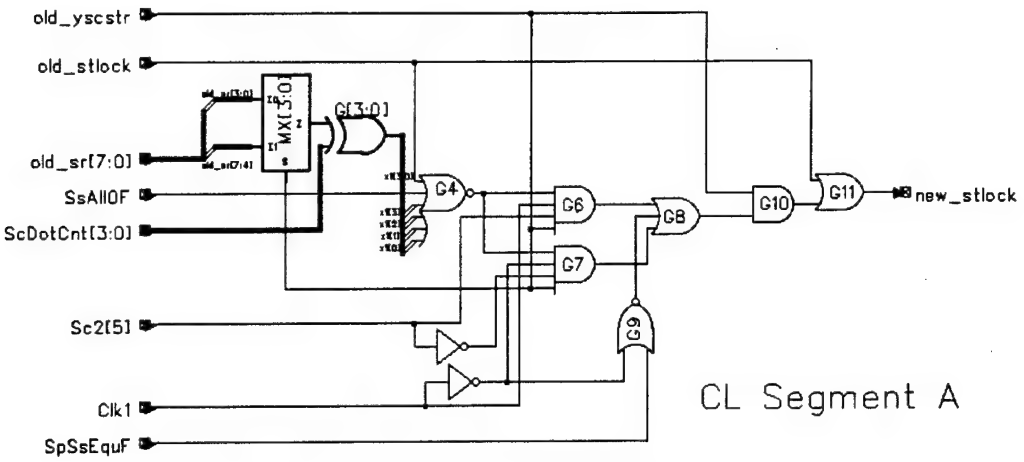
$$\text{overhead} = \frac{v - v_{\text{baseline}}}{v_{\text{baseline}}},$$

where v is the value for the circuit in question, and v_{baseline} is the corresponding value for a baseline circuit.

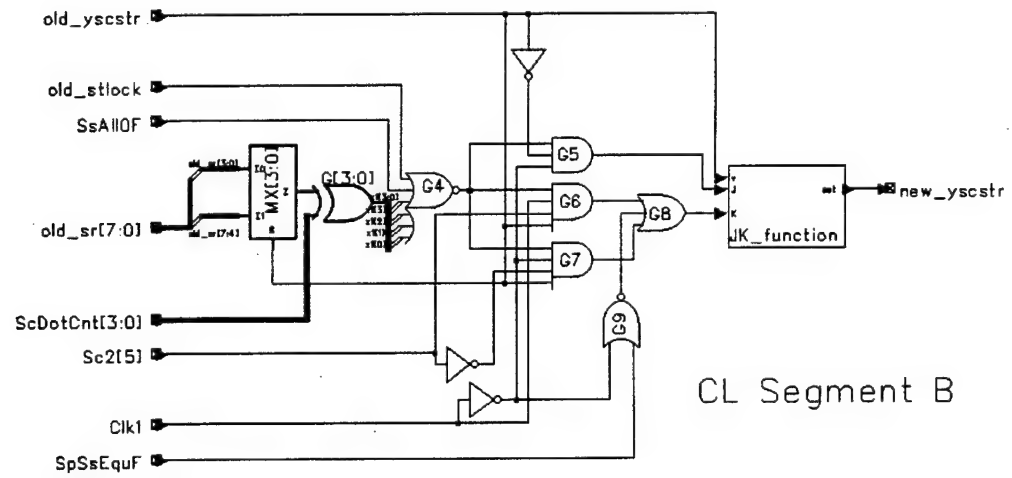
Regardless of which methodology is used for testability enhancement, the first step of the testability enhancement methodology requires calculation of all flip-flop 1-probabilities and entropies. Before this can be done, the combinational logic of the circuit must be divided into segments to create an "RTL" view of the circuit. The need for this step was described in Section 5.4 of Chapter Five, which describes the 1-probability computation. The next subsection describes the segmentation of the combinational logic for this example. A second subsection presents flip-flop entropies for the original circuit, and describes why the low entropies adversely affect testability.

7.2.1 Segmentation of combinational logic

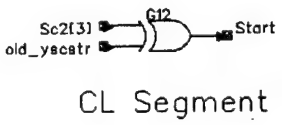
In order to create an "RTL" view of the circuit, we group the gates of the submodule into combinational logic segments driving each flip-flop and primary output. This segmentation of the circuit is shown in Figure 7-4, in which the rectangular boxes are multiplexers and the thick lines are multiple signal lines. There are two important aspects to note about the segmentation:



(a) Combinational logic driving DFF8 (signal STLOCK).



(b) Combinational logic driving JKFF0 (signal YSCSTR).



(c) Combinational logic driving primary output START.

Figure 7-4. Combinational logic of circuit, grouped into segments.

- Since our Markov Chain model handles only simple D flip-flops, we must model any JK flip-flops in terms of D flip-flops, combinational logic, and feedback, as shown in Figure 5-16 on page 93. Note that this change is for the sake of analysis only; it is not necessary to replace the JK flip-flop in the physical circuit.
- Since a gate may drive more than one flip-flop or primary output (by fanning out), the combinational logic segments may overlap. For example, OR gate *G8* is in both CL segment *A* and *B*.

The result of segmentation is a register transfer level view of the sequential CUT; this view is shown in Figure 7-5.

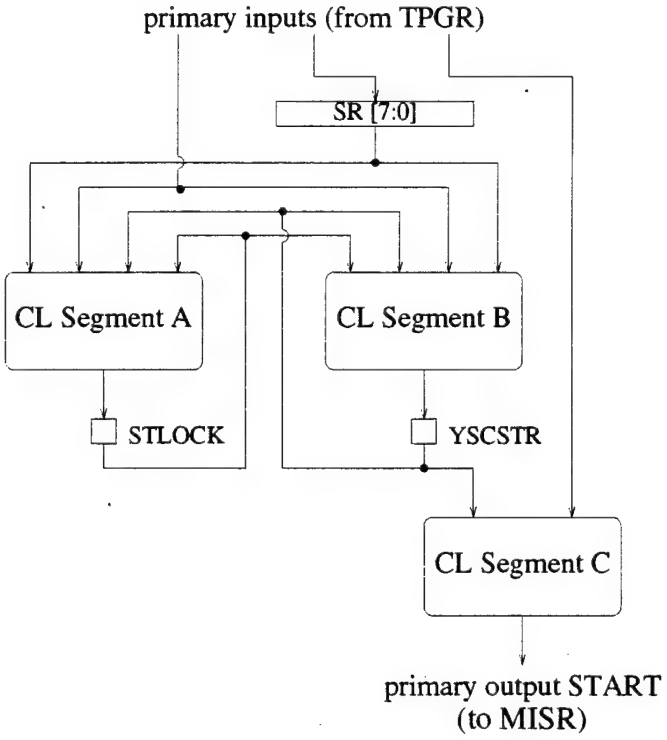


Figure 7-5. RTL view of the example circuit.

7.2.2 Analysis of flip-flop entropies

It is to the "RTL" view that we apply the Markov model to compute the flip-flop 1-probabilities and entropies. Results of the probabilistic analysis (which required two iterations) are shown in Table 7-1. These results bring two serious testability problems

signal name	1-prob.	entropy
SR [7] to SR [0]	0.5	1
STLOCK	1	0
YSCSTR	0	0
START	0.5	1

Table 7-1. Flip-flop and primary output 1-probabilities and entropies for the original circuit.

to light: both STLOCK and YSCSTR have entropies of zero in the steady-state. From Figure 7-6, which is a fragment of the original circuit affected by the STLOCK signal, it is easy to see that once the STLOCK signal goes high, feedback ensures that it stays high. Therefore, in the steady state STLOCK has a 1-probability of one, and an entropy (using Equation 4-1 on page 47) of zero. A similar problem ensuring that the YSCSTR signal stays low is less immediately apparent from the circuit structure. These low entropies cause tremendous testability difficulties; when the STLOCK signal is high, it serves to block passage of values through NOR gate *G4*, and in turn through AND gates *G5*, *G6*, and *G7* (see Figure 7-6). Thus, any fault effects observable at the inputs of these gates will not be observable at the primary output of the circuit. Thus, the STLOCK signal, when high, effectively cuts off observability of a large

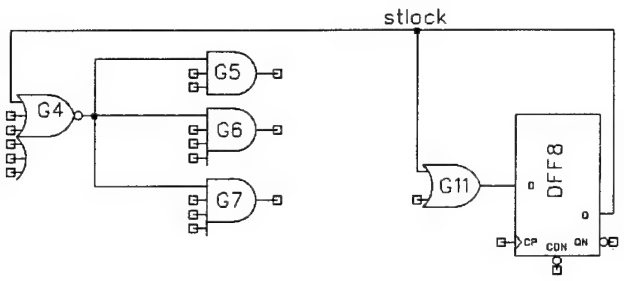


Figure 7-6. Feedback of the STLOCK signal hampers testability.

section of the circuit. The YSCSTR signal causes similar problems with AND gates *G6*, *G7*, and *G10*.

A first testability enhancement to the circuit should ensure that signals STLOCK and YSCSTR do not remain constant. The entropy of these signals can be increased in a number of ways. We will explore two ways in this chapter. Using the circular BIST methodology, we will replace the low entropy flip-flops with test flip-flops that are included in the circular BIST chain. In another experiment, we will use multiplexers to break feedback loops when the circuit is in test mode; feedback is a major cause of low entropy. A third method for enhancing flip-flop entropy is to use a partial scan chain consisting of the flip-flops with low entropy so that the states of these flip-flops can be controlled directly during the test session [LiZB93].

7.3 *Circular BIST-Based Test Point Insertion*

In this subsection, experiments that use the circular BIST methodology to enhance testability are described. The basic scheme used is illustrated in Figure 7-7. Selected flip-flops of the sequential CUT are replaced with circular BIST test flip-flops. During normal mode operation, these test flip-flops act like regular flip-flops, but in test mode they form a circular BIST chain. The first flip-flop in the chain is driven with random patterns from one output of the TPGR, and the last flip-flop in the chain drives one input of the MISR.

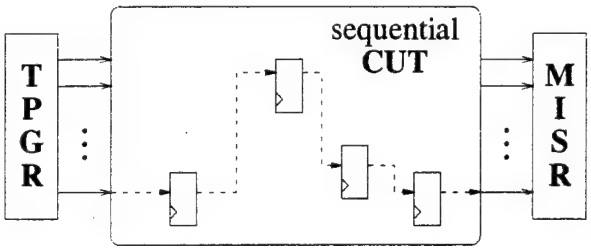


Figure 7-7. The circular BIST scheme.

We use the original circuit as a baseline circuit for the experiments described in this section. As a first enhancement (following Step 1 of Figure 7-2), flip-flop entropy is improved by including flip-flops with low entropy in the circular BIST chain. Thus, flip-flops DFF8 and JKFF0 are replaced with test flip-flops and configured into a chain. A dummy flip-flop was required between DFF8 and JKFF0 to prevent register adjacency, as described in Chapter Six. Table 7-2 compares flip-flop entropy for the original circuit, which we denote *Version I*, and the circuit variation with increased

signal name	*Version I*		*Version II*	
	1-prob.	entropy	1-prob.	entropy
SR [7] to SR [0]	0.5	1	0.5	1
STLOCK	1	0	0.5	1
YSCSTR	0	0	0.5	1
START	0.5	1	0.5	1

Table 7-2. Flip-flop and primary output 1-probabilities and entropies for the original circuit and the first circular BIST-based enhancement.

flip-flop entropy, which we denote *Version II*. From the table, we see that in *Version II* the flip-flops have very high entropy. High entropies ensure that good quality test patterns are delivered to each combinational logic segment.

The lower two curves of Figure 7-8 compare fault coverage for the original circuit and the first circular BIST variation, *Version II*. From Figure 7-8, we see that while enhancement of flip-flop entropy has improved the fault coverage for the example circuit, fault coverage is still not high. This is the case when the combinational logic seg-

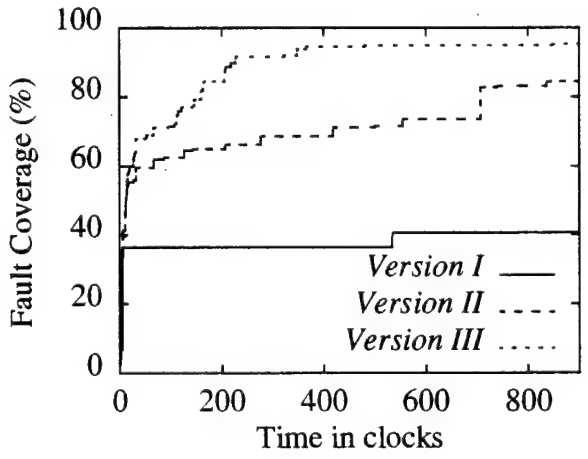


Figure 7-8. Fault coverage curves for the circular BIST-based circuits.

ments are *random pattern resistant*. One good indication of random pattern resistance is low entropy on one or more internal lines of the circuit; if a line has low entropy, one of the logic values is difficult to generate on that line. As a result, any gates that are driven by that line may be difficult to test, especially if detecting some fault near that gate involves controlling the line to the difficult-to-obtain value.

The second step of the testability enhancement technique of Figure 7-2 obtains the 1-probabilities and entropies of internal lines of the circuit by applying the Parker-McCluskey algorithm to the combinational logic segments of Figure 7-4. Results for *Version II* are shown in the first two columns of Table 7-3. These results show that even though we are supplying each combinational logic segment with uniformly distributed pseudorandom patterns, the entropy of internal lines is low. As shown in the table, the output of *G4* has an entropy of only 0.11615; this also causes low entropy in parts of the circuit driven by *G4* (*G5*, *G6*, *G7*, etc.). We can obtain a value of 1 at the output of *G4* only approximately 1/64th of the time. Since controlling this node to value 1 is necessary for fully testing gates *G5*, *G6* and *G7*, fully testing the circuit may require using an unacceptably long test session. This is reflected in the fault coverage curve for *Version II;* the curve rises slowly instead of reaching a knee and leveling off suddenly.

For our circular BIST experiment, we will improve the entropy at the output of *G4* by *inserting a test flip-flop* at this node, so that during test, gates *G5*, *G6*, and *G7* are not driven by *G4*, but are instead driven directly by the circular BIST chain. The flip-flop,

gate or node	*Version II*		*Version III*	
	1-prob.	entropy	1-prob.	entropy
MUX0 to MUX3	0.5	1	0.5	1
G0 to *G3*	0.5000	1	0.5000	1
G4 (observed)	0.1563	0.1162	0.1563	0.1162
G4 (fanouts)	—	—	0.5000	1
G5	0.003906	0.03688	0.1250	0.5436
G6	0.001953	0.02039	0.0625	0.3373
G7	0.001953	0.02039	0.0625	0.3373
G8	0.2529	0.8159	0.3125	0.8960
G9	0.2500	0.8113	0.25	0.8113
G10	0.1279	0.5517	0.1563	0.6253
G11	0.5640	0.9882	0.5781	0.9823
STLOCK	0.5000	1	0.5000	1
G12 (START)	0.5000	1	0.5000	1
JK function	0.3760	0.9552	0.3906	0.9652
YSCSTR	0.5000	1	0.5000	1

Table 7-3. Internal node 1-probabilities and entropies for the circular BIST-based circuit variations.

which is transparent in normal mode, ensures randomness at the trouble node during test. This variation of the circuit is denoted *Version III*. Table 7-3 shows that internal line entropies for *Version III* are substantially higher than for *Version II*. Figure 7-8 shows that *Version III* has significantly higher fault coverage than *Version I* and *Version II*; note that *Version III* also reaches high fault coverage much more quickly than *Version II*.

The majority of the faults not detected in *Version III* of the circuit are on the asynchronous reset lines of the flip-flops; since the reset lines are exercised only once at the

beginning of the test, it is difficult to tell whether the resets are working correctly. In particular, we may not notice if a reset is stuck in its inactive state. In the next set of experiments, which are run on exactly the same three circuits as the previous set of experiments, we remedy the situation by exercising the reset lines during the test. We do so in a pseudorandom fashion; however, we weight the pseudorandom pattern on the reset line so that resetting is done infrequently. By resetting infrequently, we can catch faults on the reset lines without disrupting the rest of the test too severely.

Figure 7-9 shows fault coverage curves for the original circuit and the two circular BIST variations, this time using pseudorandom asynchronous resets, weighted so that the probability of doing a reset at any given clock is one-sixteenth. It is important to note that this use of the asynchronous reset invalidates our entropy analysis; the flip-flop is artificially brought to the zero state, and so the probability that the flip-flop is a zero is higher. In addition, the Markov analysis works with probability distributions in the steady-state; and therefore does not apply directly after a reset.

In the case of the original circuit, the pseudorandom reset does more than allow us to catch faults on the reset lines; it also improves the testability of the rest of the circuit. It does this by kicking the circuit out of the "locked-up" state that caused the entropies to be so low (recall Figure 7-6). This is enough to bring the achievable fault coverage for the original circuit up from 41% to 72%; actually, using a pseudorandom reset is almost as good as replacing JKFF0 and DFF8 with circular BIST flip-flops. If a circuit already uses flip-flops with reset capability, this can be an area-effective method for

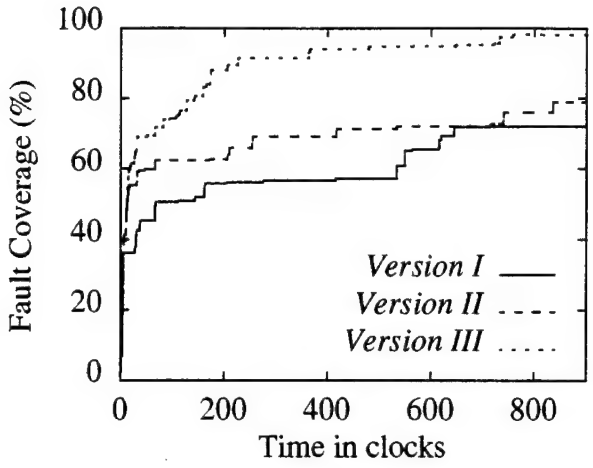


Figure 7-9. Fault coverage curves for the circular BIST-based circuits, using pseudorandom asynchronous resets.

circuit	Area in λ^2	Transistor count	Critical delay in ns
Version I	730 x 576.5	530	10.80
Version II	874 x 568.5	635	13.44
Version III	954 x 568.5	723	14.84
overhead for *Version II*	18.1%	19.8%	24.4%
overhead for *Version III*	28.9%	36.4%	37.4%

Table 7-4. Area and performance figures for the circular BIST-based circuit variations.

raising flip-flop entropy. However, using a pseudorandom reset is not enough by itself to attain high fault coverage because of the low entropy of internal lines.

Table 7-4 shows area and performance characteristics for the three circular BIST-based circuit variations. Area overhead is high, especially for *Version III*. A substantial part of the problem here is that we must insert dummy flip-flops to break up register adjacency in the circular BIST chain; this would not have been necessary if the flip-

flops had not been so interdependent on one another. The inserted test circuitry also has a detrimental effect on the critical delay of the circuit.

7.4 *Multiplexer-Based Test Point Insertion*

Instead of using circular BIST, it is possible to use multiplexer-based test point insertion for testability enhancement. Here, multiplexers are used to deliver high quality test patterns to internal parts of the circuit that would not ordinarily be controllable; the multiplexers are controlled by a signal that indicates whether the circuit is in normal or test mode.

In this section, experiments that use the multiplexer-based test point insertion to enhance testability are described. Since the design engineers suspected that observability is a major problem with the submodule of Figure 7-3, we start by taking several extra observation points out from internal parts of the circuit to the MISR. We use this submodule with added observability as the baseline circuit for the experiments in this section, and refer to it as *Version I*. For the circular BIST-based experiments described in the previous subsection, there was no need to explicitly add extra observability points to the circuit, since the flip-flops in the circular BIST chain provided observable points internal to the circuit.

As a first enhancement, flip-flop entropy is improved by breaking feedback loops during test mode. Table 7-2 compares flip-flop entropy for the baseline circuit, *Version I*,

and the circuit variation with increased flip-flop entropy. The variation, which is denoted *Version II*, uses multiplexers to break all of the feedback loops in the circuit; note that since some of the feedback was inherent in the JK flip-flop, this required replacing the JK flip-flop with a D flip-flop and appropriate combinational logic. During test, lines that are ordinarily driven by feedback are driven directly by the TPGR instead. From Table 7-2, we see that the variation has improved entropy. The higher the entropy of the inputs to a combinational logic segment, the better the quality of test patterns applied to that segment.

signal name	*Version I*		*Version II*	
	1-prob.	entropy	1-prob.	entropy
SR [7] to SR [0]	0.5	1	0.5	1
STLOCK	1	0	0.5640	0.9882
YSCSTR	0	0	0.3760	0.9552
START	0.5	1	0.5	1

Table 7-5. Flip-flop and primary output 1-probabilities and entropies for the multiplexer-based circuits.

Figure 7-10 shows the fault coverage curves for the baseline circuit, *Version I*, and the multiplexer-based variation, *Version II*. As was the case with the circular BIST experiments, we see that enhancement of flip-flop entropy improves fault coverage, but not a great deal. Again, we move to the second step of the testability enhancement technique of Figure 7-2, to take a closer look at what is happening within the combinational logic. 1-probabilities and entropies for *Version II* of the circuit are shown in the first two columns of Table 7-3. The values show that the trouble is at the same node as before; the output of *G4* has low entropy.

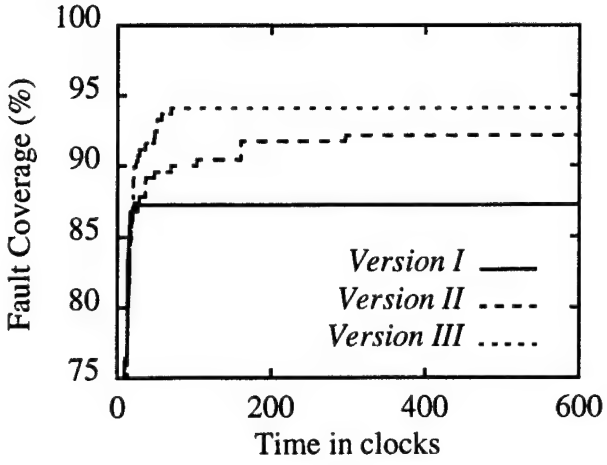


Figure 7-10. Fault coverage curves for the multiplexer-based circuits.

gate or node	*Version II*		*Version III*	
	1-prob.	entropy	1-prob.	entropy
MUX0 to MUX3	0.5000	1	0.5000	1
G0 to *G3*	0.5000	1	0.5000	1
G4 (observed)	0.1563	0.1162	0.1563	0.1162
G4 (fanouts)	—	—	0.5000	1
G5	0.003906	0.03688	0.1250	0.5436
G6	0.001953	0.02039	0.0625	0.3373
G7	0.001953	0.02039	0.0625	0.3373
G8	0.2529	0.8159	0.3438	0.9284
G9	0.25	0.8113	0.2500	0.8113
G10	0.1279	0.5517	0.2188	0.7579
G11(STLOCK)	0.5640	0.9882	0.6094	0.9652
G12 (START)	0.5000	1	0.5000	1
YSCSTR	0.3760	0.9552	0.4063	0.9745

Table 7-6. Internal node 1-probabilities and entropies for the multiplexer-based circuits.

For the multiplexer-based methodology, we will improve the entropy at the output of *G4* by *slicing* the circuit at this node; a multiplexer is inserted so that during test, gates *G5*, *G6*, and *G7* are not driven by *G4*, but are instead driven directly by the TPGR. For this approach, the output of *G4* must be connected directly to the MISR to make fault effect observation possible. This sliced variation of the circuit is designated *Version III*. Table 7-3 shows that internal line entropies for the sliced circuit are substantially higher than for the original. Figure 7-10 shows that *Version III* achieves higher fault coverage in significantly less time than *Version II*.

As was the case with the circular BIST-based variations, most of the faults remaining undetected are on the asynchronous reset lines. Fault coverage curves for the same circuits, this time using a weighted pseudorandom reset, are shown in Figure 7-10. Again, we see that if the flip-flops of a circuit already have reset capabilities, using a pseudorandom weighted reset can be an effective way to improve flip-flop entropy; in this case, the use of a pseudorandom weighted reset raised the achievable fault coverage of *Version I* up from 87% to almost 95%. As was the case with the circular-BIST based circuit variations, however, a pseudorandom weighted reset alone is not enough to achieve high fault coverage, because it can not remove the random pattern resistance that causes lines internal to the combinational logic to have low entropy.

Comparisons of area and performance for the three multiplexer-based circuit variations are given in Table 7-7. By comparing these numbers to the results for the circular BIST experiment, we see that the multiplexer-based test point insertion requires much

less area overhead to do the job. However, the critical delay overhead for *Version III* is a quite substantial 22.8%. The reason for this is that the inserted multiplexers lie on the critical path.

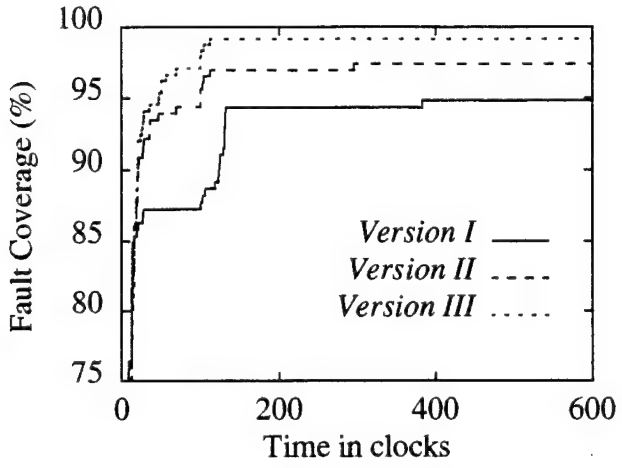


Figure 7-11. Fault coverage for the multiplexer-based circuits, using a pseudorandom asynchronous reset.

circuit	Area in λ^2	Transistor count	Critical delay in ns
Version I	746 x 560.5	530	10.81
Version II	770 x 584.5	575	12.20
Version III	810 x 584.5	589	13.28
overhead for *Version II*	7.6%	8.5%	12.9%
overhead for *Version III*	13.2%	11.1%	22.8%

Table 7-7. Area and performance figures for the multiplexer-based variations.

Work by [LiZB93] points out that it is possible to use gates rather than full multiplexers to do test point insertion. For our example, we could use an OR gate to slice the circuit at the output of gate *G4*. One input of the OR gate is driven by the output of *G4*, and the other is driven by the TPGR. During test mode, the output of *G4* almost

always has value zero, so the output of the inserted OR gate has entropy nearly equal to one. During normal mode, we must be careful that the TPGR generates a zero, so that the inserted OR gate does not affect the functioning of the circuit. Lin's test point insertion will have substantially less impact on area and performance, since an OR gate is both smaller and faster than a full multiplexer.

7.5 *Summary*

Many existing circuit designs are not sufficiently testable; they require testability enhancement, either through complete re-synthesis or small, local design modifications. When re-synthesis of a design can not be done, the only alternative is to try to enhance the testability of the existing design. The proposed technique for probabilistic analysis of sequential gate-level circuits provides a systematic approach for testability enhancement of sequential circuits. It has been used to enhance the testability of a sample submodule of an industrial design. For the style of design used, i.e., for relatively small sequential circuits interconnected via system registers, the technique is computationally inexpensive, and helps to quickly find areas of random pattern resistance in the design. These problem areas can then be eliminated with simple design modifications. Resulting modified designs enjoy significantly higher fault coverage than the original.

Chapter 8

BIST Insertion at the Register Transfer Level

Traditionally, BIST insertion at the register transfer level is done by making each arithmetic logic unit (ALU) in a datapath directly testable; controllable registers (TPGRs) are placed at each ALU's inputs, and an observable register (MISR) is placed at each ALU's output. However, such an invasive addition of test registers may not be necessary. For example, suppose that the input registers to an ALU are not directly controllable, but that they still generate patterns that are random enough to effectively test the ALU. In this case, there is no need to replace the normal system registers with test pattern generation registers; actually, to do so is often undesirable, since TPGRs are larger and slower than ordinary registers. Thus, in selecting registers for BIST insertion, a trade-off can be made, with test quality on one side and area and performance on the other. This chapter explores this trade-off for register transfer level datapaths.

8.1 Experimental Set-Up

In this chapter, experiments are done with several register transfer level circuits. Three testability metrics are used: randomness, expected state coverage, and transparency. The testability metrics are computed for the registers of the circuits using the Markov model of Chapter Five, and decisions about where to place test registers within the circuits are based on the metrics. Fault coverage curves show the effects of the BIST insertion decisions on the testability of the resulting circuits. These experiments serve two purposes: *to validate the Markov model* by showing that the analytical predictions are close to actual values obtained by simulation, and t*o show how the metrics can be used* to compare different BIST configurations in terms of test quality, and therefore to guide BIST insertion.

Analytical predictions for the testability metrics are from the Markov model; computation times for the Markov model analysis are for a SPARCstation IPC with 36 MB of memory. Actual values for the testability metrics are obtained from a computer program that simulates the functioning of an RTL circuit while in test mode, computing the registers' states at each time step. The simulation uses a system state seed of all '1' bits. The choice of system state seed is arbitrary; any seed that does not lead too quickly to a system state cycle will produce similar results.

Layout area, transistor count, and critical delay figures are given so that the trade-off between test quality and cost can be better understood. All are derived in the COM-

PASS Design Automation suite of tools [CODA92]. Layout area is found by placing and routing the circuit. We include transistor count as a supplementary measure of area, because layout area depends heavily on the algorithm used for the routing and the manufacturing process used. For example, layout area results here are for a process with two levels of metallization; if three or four levels are available, the layout areas will be considerably different. The critical delay for the circuit is the delay along the slowest combinational path in the circuit, and reflects the speed at which the circuit can be clocked. It includes both gate delays and interconnect delays based on the routing. For all area and performance figures, overhead is given in terms of a version of the circuit which contains the minimal amount of hardware needed for the circular self-test path technique; overhead for a circuit is calculated with the following formula:

$$\text{overhead} = \frac{v - v_{\min}}{v_{\min}},$$

where v is the value for the circuit in question, and $v_{\min}$ is the corresponding value for the "minimal" circuit.

Fault coverage curves are from GENTEST, an AT&T fault simulator, and include only those faults within the ALUs. The test session for the circuit begins by configuring the circular BIST test registers into a scan chain, and shifting in the desired system state seed. This initialization phase is not included in the fault coverage curves. Traditionally, the signature is obtained by observing one element of the circular self-test path

for a limited number of clocks at the end of the test session. However, we found this difficult to implement in GENTEST; instead, we observed the element throughout the entire test session. As a result, our experiments neglect some of the probability of aliasing during circuit response compaction.

In these examples, the values of the testability metrics are used to decide where to insert test registers. Low randomness can have an adverse effect on test quality. At best, low randomness values require the use of a longer test session to adequately test the ALUs in the circuit; in some cases, however, even an increased test length will not help to generate an adequate number of different test patterns. Expected state coverage can help distinguish between these two cases. Low transparency can also adversely affect test quality by making it difficult to observe the effect of faults, as discussed in Chapter Four.

8.2 *Example One: A Cascade*

The first example, shown in Figure 8-1, is a four bit wide cascade of adders and multipliers that performs the arithmetic function $F = (ab)\ (cd) + (ef)\ (g + h)$. Testability metrics, both from analysis and simulation, are shown in Table 8-1 for two different circular self-test path choices. Part (a) of the table corresponds to a version of the circuit with a *minimal* circular self-test path, that is, with a path containing only the primary input and primary output registers; for this circuit, which we refer to as *Version I*,

the circular self-test path is ($\Leftarrow$ REG8, $\Leftarrow$ REG7, $\Leftarrow$ REG6, $\Leftarrow$ REG5, $\Leftarrow$ REG4, $\Leftarrow$ REG3, $\Leftarrow$ REG2, $\Leftarrow$ REG1, $\Leftarrow$ REG15), where the order shown indicates the order of the registers in the path, and $\Leftarrow$ denotes that the indicated register is shifted to the left, from least significant bit to most significant bit.[1] [2] Randomness (MR), expected state coverage (ESC) in 96 clocks, and transparency (MT) of all registers are shown.

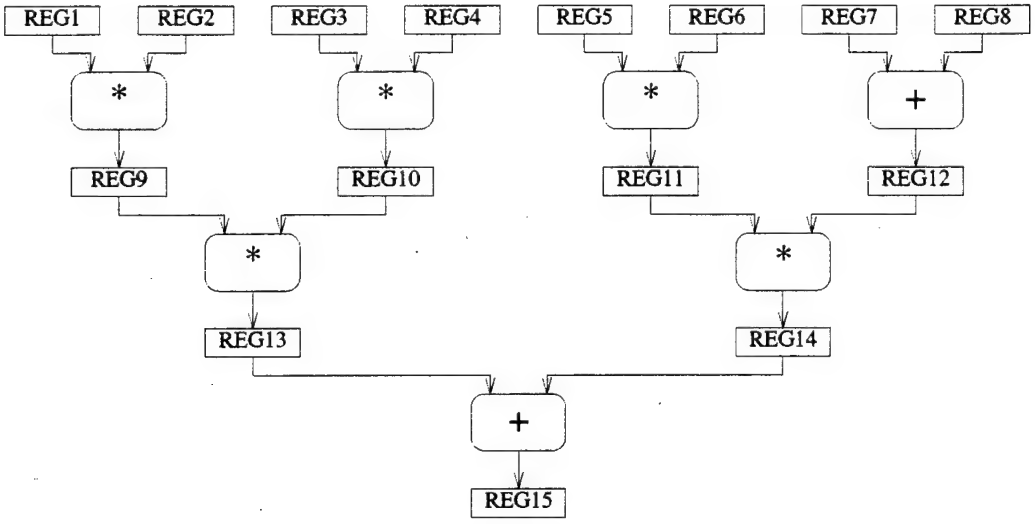


Figure 8-1. A cascade of adders and multipliers.

For this circuit with a minimal circular self-test path, the testability metrics of some registers not in the path is less than ideal. In particular, registers REG13 and REG14 have low randomness, and registers REG9, REG10, REG11, and REG12 have low

1. We will later use $\Rightarrow$ to denote that a register is shifted to the right.
2. For each example in this chapter, the order of the registers in the circular self-test path, and the orientation or shift direction for each register, were chosen to avoid bit correlation problems as described in Chapter Six.

		REG1-8	REG9	REG10	REG11	REG12	REG13	REG14	REG15
MR	Analy.	1	0.9295	0.9295	0.9295	1	0.6389	0.7915	1
	Sim.	0.9999	0.9298	0.9296	0.9295	0.9996	0.6400	0.7912	0.9997
ESC in 96 clks	Analy.	0.9981	0.9756	0.9756	0.9756	0.9981	0.7525	0.8875	0.9981
	Sim.	1	0.9375	0.9375	1	1	0.7500	1	1
MT	Analy.	1	0.7000	0.7000	0.8670	0.7000	1	1	1

(a) *Version I*, with a minimal circular self-test path.

MR	Analy.	1	1	1	1	1	0.9295	0.9295	1
	Sim.	0.9999	0.9999	0.9999	0.9999	0.9999	0.9308	0.9286	0.9999
ESC in 96 clks	Analy.	0.9981	0.9981	0.9981	0.9981	0.9981	0.9756	0.9756	0.9981
	Sim.	1	1	1	1	1	1	1	1
MT	Analy.	1	1	1	1	1	1	1	1

(b) *Version II*, with registers REG9-12 added to the path.

Table 8-1. Testability metrics for the cascade.

transparency. In order to boost these low values, we choose to insert additional BIST registers in place of registers REG9, REG10, REG11, and REG12; for *Version II* of the circuit, the circular self-test path is ($\Leftarrow$REG8, $\Leftarrow$REG7, $\Leftarrow$REG6, $\Leftarrow$REG5, $\Leftarrow$REG4, $\Leftarrow$REG3, $\Leftarrow$REG2, $\Leftarrow$REG1, $\Leftarrow$REG15, $\Leftarrow$REG12, $\Leftarrow$REG11, $\Leftarrow$REG10, $\Leftarrow$REG9). Testability metrics for this variation are in part (b) of Table 8-1, and are all improved.

For the circular BIST methodologies, the insertion of a test register boosts both controllability and observability; that is why our primary inputs have high transparency. Note also that although we replaced registers 9 and 10 with test registers for the purpose of improving transparency, it also brought the randomness of these registers up to 1. This in turn was sufficient to bring the randomness of register 13 up from a low

value of 0.6389 to the more reasonable value of 0.9295, and the randomness of register 14 up from 0.7915 to 0.9295.

Table 8-1 shows a good agreement between the analytical values predicted by our Markov model and actual values obtained from simulation. In general, we expect a closer agreement for randomness than for expected state coverage, because actual values for expected state coverage depend heavily on the choice of the system state seed for short test sessions. Computation of the testability metrics for both circuits using the Markov model required a total of 1.03 CPU seconds.

Figure 8-2 shows fault coverage curves for the two versions of the circuit. For this example, there is not a large difference in the attainable fault coverage; *Version I* leaves only one fault undetected, and *Version II* leaves no faults undetected. However, the curves show that the *Version II* is significantly more testable than *Version I* in the sense that it reaches high fault coverage much more quickly. *Version II* reaches 100% fault coverage in 105 clocks; in contrast, *Version I* requires almost three times as much time, 300 clocks, to reach its top fault coverage of 99.9%.

Table 8-2 shows the overhead involved in adding the new test registers to the circuit. It shows area and performance figures for the two versions of the cascade. From the table, we see that changing registers 9 through 12 into circular BIST test registers resulted in a circuit that was less than 9% bigger than the original. The addition also slowed the circuit down 20.8%.

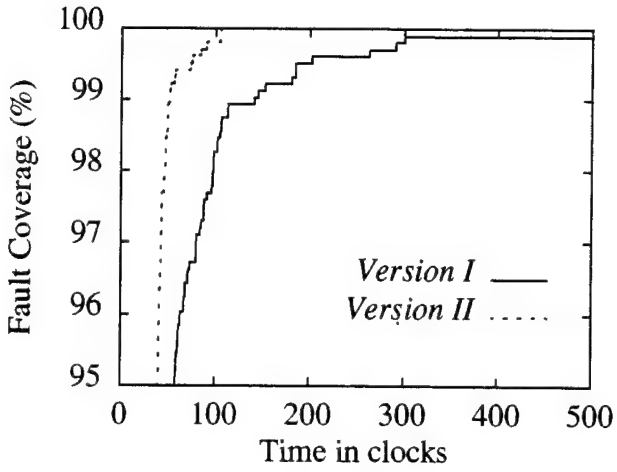


Figure 8-2. Fault coverage curves for the cascade.

	Area in λ^2	Transistor Count	Critical Delay in ns
Version I	2242 x 1407	4140	10.72
Version II	2346 x 1415	4492	12.95
overhead	5.2%	8.5%	20.8%

Table 8-2. Area and performance figures for the cascade.

In practice, a trade-off can be done between the test quality and the impact on area and performance. In this example, we could have chosen to insert fewer test points; for example, we may have decided to replace registers 9, 10, and 12 with circular BIST registers to boost their transparencies of 0.7000 up to 1, but to leave register 11, with a transparency of 0.8670, a normal register. This decision would have had less of an impact, particularly on area, and would also have resulted in a circuit somewhere between *Version I* and *Version II* in terms of test time required to achieve high fault coverage.

8.3 *Example Two: An Arithmetic Pipeline*

Our second example, shown in Figure 8-3, is typical of an arithmetic pipeline, and is four bits wide. It computes the recurrence relation $F(t+3) = \frac{a(t)}{b(t)} + c(t)\,F(t+1)$, where a, b, and c are primary input values. Testability metrics for the pipeline are shown in Table 8-3. Again, the first version has a minimal circular self-test path; in this case, the path is ($\Leftarrow$REG3, $\Leftarrow$REG2, $\Leftarrow$REG1, $\Leftarrow$REG7). *Version I* has good transparency throughout; this is because the use of the circular self-test path technique adds transparency to the inputs of the circuit, and the high sensitivity of the adder to changes in its input makes the middle of the circuit highly transparent. One register of the circuit, REG4, has an extremely low randomness value of 0.5883. *Version II* of the circuit removes this problem by placing REG4 in the circular self-test path ($\Leftarrow$REG3, $\Leftarrow$REG2, $\Leftarrow$REG1, $\Rightarrow$REG4, $\Leftarrow$REG7). Computation of the testability metrics for both circuits using the Markov model required a total of 1.44 CPU seconds.

Fault coverage curves for the two versions, given in Figure 8-4, show that *Version II*'s addition of a single test register causes a dramatic improvement in fault coverage; attainable fault coverage goes up from 91.8% in *Version I* to 99.8% in *Version II*. Table 8-4, which gives area and performance figures for the two versions of the pipeline, shows that the price paid for the added testability is a less than 7% increase in area and an 8.5% increase in critical delay.

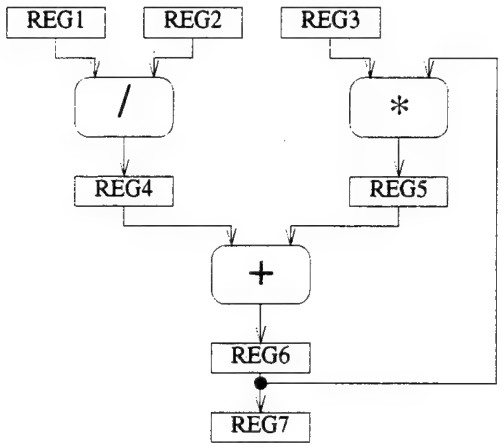


Figure 8-3. An arithmetic pipeline.

		REG1-3	REG4	REG5	REG6	REG7
MR	Analysis	1	0.5883	0.8854	0.9771	1
	Simulation	0.9929-0.9930	0.5783	0.8835	0.9762	0.9927
ESC (in 96 clocks)	Analysis	0.9981	0.6381	0.9650	0.9944	0.9981
	Simulation	1	0.5000	0.9375	1	1
MT	Analysis	1	1	1	1	1

(a) *Version I*, with a minimal circular self-test path.

MR	Analysis	1	1	0.9295	1	1
	Simulation	0.9996	0.9980	0.9336	0.9967	0.9989
ESC (in 96 clocks)	Analysis	0.9981	0.9981	0.9756	0.9981	0.9981
	Simulation	1	1	1	1	1
MT	Analysis	1	1	1	1	1

(b) *Version II*, with REG4 added to the path.

Table 8-3. Testability metrics for the arithmetic pipeline.

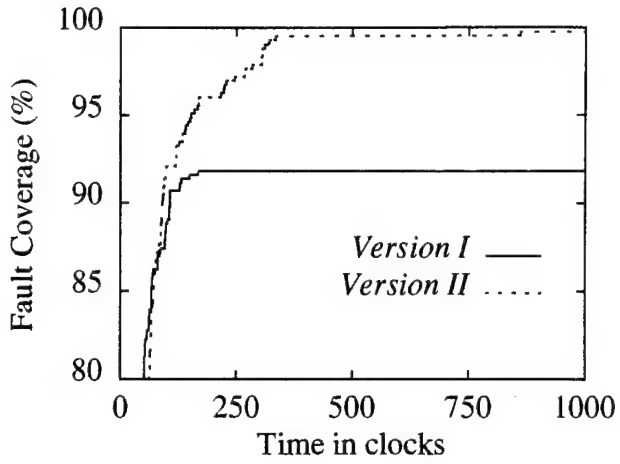


Figure 8-4. Fault coverage curves for the arithmetic pipeline.

	Area in λ^2	Transistor Count	Critical Delay in ns
Version I	1602 x 969.5	1944	10.61
Version II	1650 x 1001.5	2032	11.51
overhead	6.4%	4.5%	8.5%

Table 8-4. Area and performance figures for the arithmetic pipeline.

8.4 *Example Three: A Low Pass Filter*

Our third example, shown in Figure 8-5, is adapted from a low pass filter in [COOS93]. The datapath is nine bits wide, using fixed point numbers with two bits after the binary point. The 25% and 75% ALUs multiply their single inputs by 0.25 and 0.75, respectively.

Version I of the circuit has a minimal circular self-test path. Testability metrics are shown in Table 8-5. Because this circuit has chained ALUs, we calculate metrics not

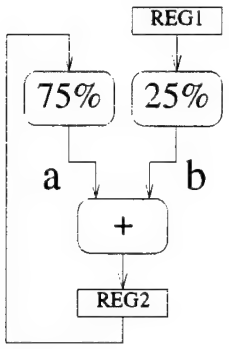


Figure 8-5. A low pass filter.

only at the registers, but also at points *a* and *b*.[3] Computation of the testability metrics using the Markov model required a total of 3.50 CPU seconds. For this example, transparency values are trivially very high, because addition is highly sensitive to changes in an input, but the randomness of point *b* is quite low because the multiplication by $\frac{1}{4}$ clears the two most significant bits. In order to improve the randomness at *b*, we insert a test register that is transparent in normal mode. *Version II* of the circuit inserts a full nine bit test register at point *b*. For this example only two of the nine bits actually have low entropy, so we can save area by using a test register of only two bits; we do this in *Version III*. Fault coverage curves for all versions of the circuit are shown in Figure 8-6. Note that *Versions II* and *III* are virtually indistinguishable in terms of fault coverage; in terms of test quality, enhancing only the low entropy bits of point *b* is sufficient. Table 8-6 shows the area and performance figures for all three versions of the low pass filter; from the figures, it is clearly beneficial to use the smaller, two bit test register at point *b*, since the area overhead is much smaller for *Version III* than it is for

3. This is accomplished by creating a version of the low pass filter that has "pseudo-registers" at points *a* and *b*, and applying the Markov model to that version in the usual way.

Version II. The performance overhead for *Version III* is a bit larger for than for *Version II*. The difference is small, and due to the simulated annealing algorithm used by COMPASS to do routing; there is an element of randomness in the algorithm, so if the routing were done a second time, the performance numbers might easily be reversed.

		REG1	REG2	*a*	*b*
MR	Analysis	1	1 .	0.9444	0.7778
	Simulation	0.9860	0.9870	0.9351	0.7750
ESC (in 3072 clocks)	Analysis	0.9975	0.9975	0.7488	0.2500
	Simulation	0.9961	0.9980	0.7500	0.2520
MT	Analysis	1	1	1	1

(a) *Version I*, with a minimal circular self-test path.

MR	Analysis	1	1	0.9444	1
	Simulation	0.9850	0.9855	0.9345	0.9878
ESC (in 3072 clocks)	Analysis	0.9975	0.9975	0.7488	0.9975
	Simulation	1.000	0.9941	0.7500	1.000
MT	Analysis	1	1	1	1

(b) *Versions II* and *III*, with a test register at *b*.

Table 8-5. Testability metrics for the low pass filter.

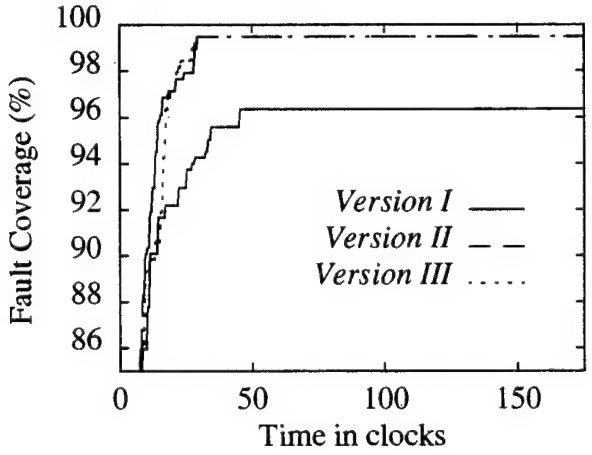


Figure 8-6. Fault coverage curves for the low pass filter.

	Area in λ^2	Transistor Count	Critical Delay in ns
Version I	1402 x 725	1506	17.88
Version II	1666 x 857.5	2190	18.05
Version III	1522 x 765	1658	18.12
overhead for *Version II*	40.5%	45.4%	1.0%
overhead for *Version III*	14.5%	10.1%	1.3%

Table 8-6. Area and performance figures for the low pass filter.

8.5 *Example Four: An Elliptical Wave Filter*

Our final example is derived from a *fifth order elliptical wave filter* commonly used as a benchmark for high level synthesis systems [PaKn89]. First, a four bit wide wave filter was synthesized without test features using the SYNTEST high level synthesis system [HPCN92]. Next, since our test effort focuses on the arithmetic logic units (ALUs), the assumption that multiplexer control would be held constant during testing was made, and so a single path through each multiplexer was chosen. In effect, this allowed the removal of all multiplexers and some registers not used for testing from the circuit. The resulting circuit is shown in Figure 8-7, and contains reconvergent fanout, feedback loops, and self-adjacency. Next, circular BIST features were added to the circuit. For this example, a minimal circular self-test path was added: ($\Leftarrow$REG10, $\Leftarrow$REG11, $\Leftarrow$REG12, $\Leftarrow$REG13, $\Leftarrow$REG4). Testability metrics are shown in Table 8-7. Computation of the testability metrics using the Markov model required a total of 3.09 CPU seconds. In this case, the randomness and transparency values for all the registers are quite high. On the basis of the high testability metrics, it

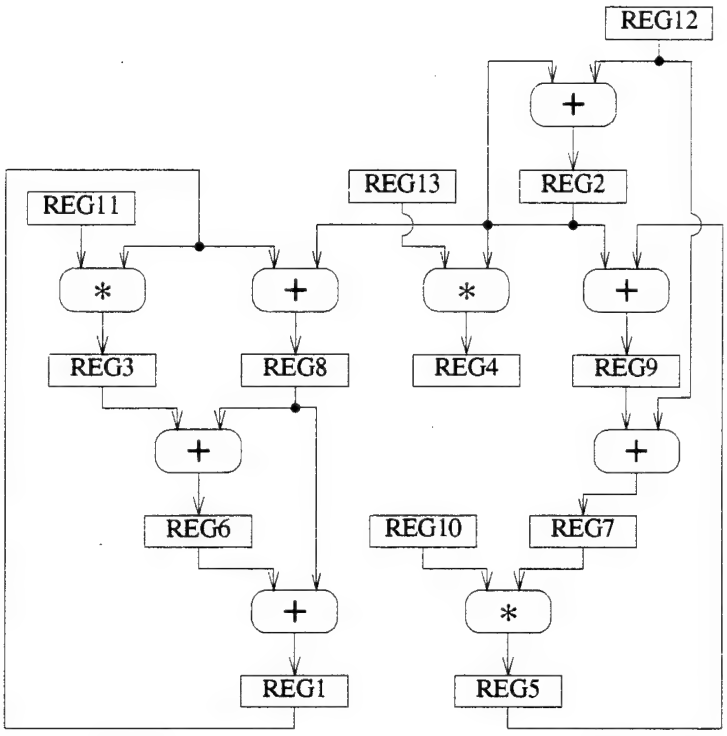


Figure 8-7. An example derived from an elliptical wave filter.

was decided that it was not necessary to add test features to any more of the registers. A fault coverage curve for the circuit, given in Figure 8-8, shows that 100% fault coverage is quickly reached for the circuit with a minimal circular self-test path. Area and performance figures are given in Table 8-8.

Note that different choices of multiplexer control result in different testability metrics for the registers of the circuit; by choosing a path through each multiplexer, we are configuring the circuit in a specific way during test. The configuration chosen here required only a minimal circular self-test path, but another configuration may require additional test registers, resulting in a larger and slower circuit. A further direction of

		REG1	REG2	REG3	REG4	REG5	REG6	REG7	REG8	REG9	REGS 10-13
MR	Analysis	1	1	0.9295	1	0.9295	1	1	1	1	1
	Simulation	0.9998	0.9998	0.9268	0.9998	0.9303	0.9998	0.9998	0.9998	0.9999	0.9999 -1.000
ESC (in 96 clks)	Analysis	0.9981	0.9981	0.9756	0.9981	0.9756	0.9981	0.9981	0.9981	0.9981	0.9981
	Simulation	1	1	1	1	1	1	1	1	1	1
MT	Analysis	1	0.8667	1	1	1	1	1	0.9333	1	1

Table 8-7. Testability metrics for the wave filter example.

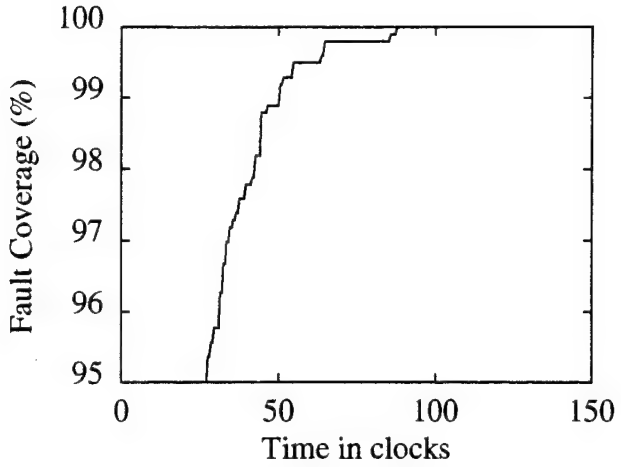


Figure 8-8. Fault coverage curve for the wave filter example.

Area in λ^2	Transistor Count	Critical Delay in ns
2186 x 1282.5	3600	12.75

Table 8-8. Area and performance figures for the wave filter example.

research might involve how to choose paths through the multiplexers so as to make the ALUs of the circuit testable with a minimum of area and performance overhead.

8.6 *Summary*

The examples of this chapter show how testability metrics can be used to guide built-in self-test (BIST) insertion in register transfer level datapaths. The testability metrics of randomness, expected state coverage, and transparency are used to implicitly capture the properties of a successful built-in self-test. Fault coverage results show that

BIST insertion done on the basis of these metrics significantly eases the testability problem without resorting to an expensive traditional BIST implementation, which requires that test registers be placed around each arithmetic logic unit.

Chapter 9

BIST Insertion in the Behavioral Domain

As the complexity of very large scale integration (VLSI) design grows, design synthesis tools have been developed at higher and higher levels of design abstraction. Over the last ten years or so, there are have been many projects to build *high level synthesis tools* that take as input a behavior, or a description of the function to be performed by the circuit in algorithm form, and produce a structure, usually a register transfer level (RTL) implementation of the behavior.

Although design synthesis tools continue to move to higher levels of design abstraction, design-for-testability (DFT) tools have not kept pace. In Chapter Three, we outlined a number of current projects dealing with testability insertion in high level synthesis, based on both BIST and automatic test pattern generation (ATPG). These approaches are all limited in some way. Most incorporate testability by imposing their own restricted design style; for example, the work of [PaCH91] enhances BIST testability by removing self-loops in the datapath, and the work of [DePo94] enhances ATPG testability by minimizing the number of large feedback loops. Most current high level synthesis for testability approaches lack the flexibility to coordinate with

general purpose synthesis tools, and many lack testability analysis capability to guide the synthesis process.

This chapter presents a new method for test synthesis in the behavioral domain based on BIST. Specifically, we have developed a technique for BIST insertion in behavioral design descriptions that coordinates through a common VHSIC hardware description language (VHDL) interface with other synthesis tools in the behavioral and structural domains. Our approach operates on a given design behavior, expressed in behavioral VHDL, that describes the desired behavior of the design during normal mode operation. The basis of our approach is to derive a test behavior from the design behavior. The test behavior, which is also expressed in VHDL, describes the behavior of the design in test mode, and is derived by fixing any testability problems in the design behavior. Thus, the test behavior describes in essence the BIST insertion for the design. A merging of the normal-mode design behavior and the test-mode test behavior is then synthesized to produce a testable design with inserted BIST structures.

One key aspect of our approach is the use of behavioral testability metrics that quantify the controllability and observability of signals embedded within a behavior. By using these metrics to quantify the testability of behaviors, we can modify behaviors before design synthesis even begins so as to ensure that the resulting circuit, when synthesized by a general purpose high level synthesis tool, will be easily testable using a simple BIST scheme. It should be noted that our behavioral insertion can employ not

only our own test metrics but also test metrics developed by others, for example, those of [PeKu94], based on flowgraph analysis of a behavior.

Another key aspect of our research has involved exploration of the connection between behavioral testability and testability of physical implementations of the behavior. By showing what ramifications a low testability signal in the behavior can have on testability at the gate level once the behavior is synthesized, we provide a link between our behavioral testability metrics and structural testability of the synthesized circuit.

Our approach goes beyond current approaches that focus exclusively on the datapath by developing a methodology that produces circuits that are completely testable, both datapath and controller together. Behavioral insertion is used to enhance the testability of the datapath, while structural insertion is used for the controller. The fact that our approach is designed to be used with any general purpose high level synthesis system adds flexibility to our approach.

The chapter is organized as follows. Section 9.1 describes the minimal behavioral BIST scheme for datapaths, and explains the rationale behind the scheme. Section 9.2 shows how our testability metrics are used as the basis of a test point insertion methodology in the behavioral domain to improve the testability of the datapath. Section 9.3 describes the structural test point insertion that we use to improve the testability of the controller and the interface between the datapath and the controller. Section 9.4

describes our overall test scheme, which uses the behavioral test scheme to test the datapath and a circular BIST scheme to test the controller. Section 9.6, Section 9.7, and Section 9.8 provide results for three example circuits, and Section 9.9 is a summary.

9.1 A Behavioral Test Scheme

Conventional BIST schemes for datapaths partition the datapath into small kernels, and test each kernel separately. The simplest conventional test scheme breaks the datapath up so that each kernel contains a single functional unit (ALU). Test pattern generation registers (TPGRs) are placed at the inputs of each kernel, and multiple input shift registers (MISRs) are placed at the outputs of each kernel, so that each ALU is tested separately. Test scheduling determines which kernels can be tested in parallel.

Many deterministic or automatic test pattern generation (ATPG) based approaches also rely on partitioning of the datapath. For example, the work of [BhJh94] uses behavioral information to find a *test environment* for each ALU in a datapath individually. By setting the control in a special way so as to allow test vectors to be justified at the inputs to the ALU and test responses to be propagated from the output of the ALU to the primary outputs, this method in effect separates the ALU from its behavioral environment during test.

In contrast, our approach tests the datapath as a whole; our test scheme, which we call *minimal behavioral BIST*, is shown in Figure 9-1 as viewed in both the behavioral and structural domains. In the behavioral domain, we separate the signals and variables of a behavior into three disjoint subsets: *input* signals, *output* signals, and *internal* variables. Our scheme makes all input signals directly controllable by mapping them onto BIST input registers in the register transfer level (RTL) circuit implementation of the behavior, and makes all output signals directly observable by mapping them onto BIST output registers in the RTL. The scheme is *minimal* in the sense that no insertion is done within the behavior; no internal variables are made BIST. In the structural domain, this means that the datapath is tested as a whole by placing a TPGR across all the primary inputs (PIs) to the datapath, and an MISR across all the primary outputs (POs). During test mode, as the TPGR generates test patterns, the datapath is exercised according to its behavior for which it was synthesized.

An important consequence of our behavior-based scheme in the structural domain is that we exercise every single connection and component in the circuit. This approach is in contrast to traditional BIST approaches; in approaches that focus on the arithmetic logic units, some registers and multiplexers may not be part of any kernel, so that some components may be neglected. In addition, interconnections between kernels may be neglected. A similar limitation exists for the ATPG-based work of [BhJh94]; concentrating on the arithmetic logic units does not guarantee high fault coverage for the overall datapath, so if high enough fault coverage is not obtained, the

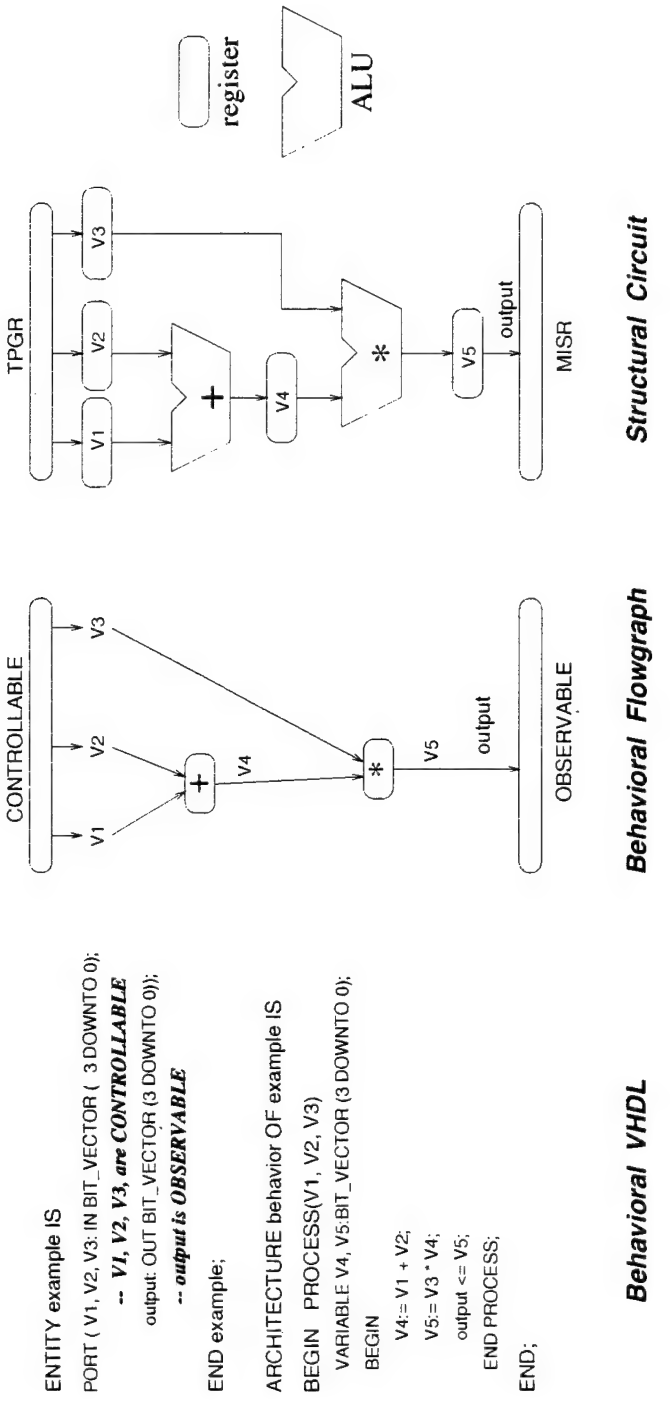


Figure 9-1. Behavioral and structural views of minimal behavioral BIST scheme.

method must add an extra step that focuses on the registers or multiplexers that were missed. By exercising the datapath according to the behavior, we can guarantee that *all* parts of our datapath will be thoroughly exercised, because each control mode of each component is used at some point during the behavior. For example, if the datapath contains a three-to-one multiplexer, each of the three paths through the multiplexer will be used at some point if the datapath is exercised according to the behavior. If one of the paths were not exercised by the behavior, then a three-to-one multiplexer would not have been synthesized in the first place; a two-to-one multiplexer would have been used instead.

Exercising the datapath circuit according to the behavior for which it was synthesized even while testing is intuitively appealing for a number of reasons. The first is that we eliminate the difficult question of how to partition the circuit into kernels, and how to schedule the testing of the kernels. Secondly, the control signals that we need to test our datapath are almost exactly those control signals that we use for our datapath in normal mode. This means that our test controller is relatively simple, and easily embedded within the system controller with a minimum of area overhead.

Note that some special care must be taken in the design of the components to be sure that each component is testable using only its defined control modes. For example, suppose that a three-to-one multiplexer is implemented with two control mode bits. The multiplexer has three control modes, but four possible combinations of control

signals. The multiplexer must be designed such that the fourth, undefined control mode is not needed to test the multiplexer.

The minimal behavioral BIST scheme will yield high fault coverage for a datapath only if the datapath can be successfully tested by applying pseudorandom patterns at its primary data inputs. Behavioral testability insertion is used to modify the design behavior so that the synthesized datapath will meet this requirement, regardless of the high level synthesis tool used. In order to describe the testability of signals embedded within a behavior, we use the randomness and transparency metrics defined in Chapter Four. In the next section, we describe our procedure for behavioral testability insertion.

9.2 *Behavioral Testability Insertion*

This section describes the use of testability metrics in the behavioral domain to pinpoint and correct testability problems in behaviors.

9.2.1 Basic concepts

We do testability insertion in behaviors by means of a new concept that we call the *test behavior.* This concept is based on the addition of a test mode in the behavioral domain that is generally different from the normal mode behavior, or *design behavior.* The test behavior can be generated by applying a series of local behavioral-for-test

transformations to the design behavior. These transformations or modifications are driven by our test metrics with the aim to enhance the testability qualities of the behavior.

The modified behavior is capable of executing both the design behavior and the test behavior. The concept is pictured in Figure 9-2. A behavioral switch determines which of the two behaviors is executed. Since the modified behavior is a merging of the design and test behaviors, we call it the *design-and-test behavior.* When the design-and-test behavior is synthesized, the result is a circuit that can be run either in normal mode, or in a mode more conducive to test. Thus, the resulting structure does not have testability problems.

In general, behavioral-for-test transformations can produce test behaviors that bear little resemblance to the design behavior. For example, a test behavior, after it is synthe-

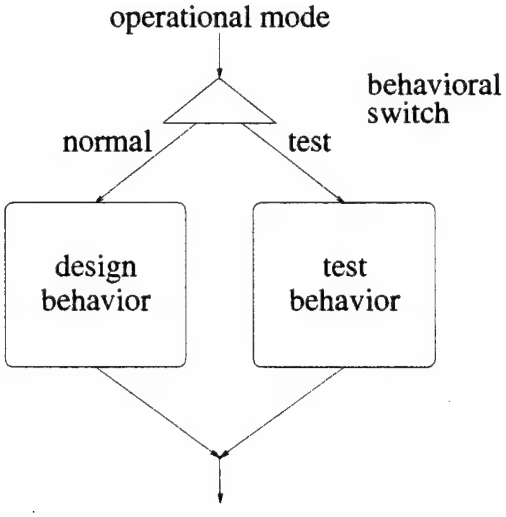


Figure 9-2. The design-and-test behavior concept.

sized, may target the testing of all ALUs of the circuit structure in parallel by reconfiguring all registers as BIST in test mode. We have decided to restrict our transformations such that the test behaviors generated are very close to the design behavior. The rationale is that such test behaviors are easily generated and easily synthesized with the design behavior in a unified way by a synthesis system. Furthermore, with this approach the test controller is easily embedded in the system controller.

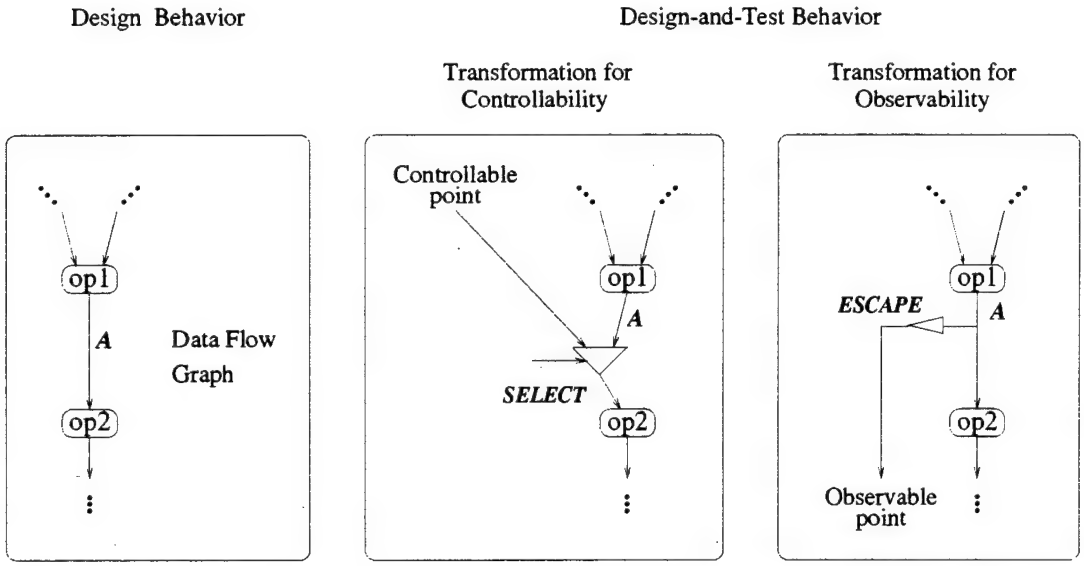


Figure 9-3. Transformations for the behavioral test scheme.

Two typical behavioral-for-test transformations are shown in Figure 9-3. SELECT and ESCAPE serve to enhance the controllability and observability, respectively, of internal behavioral variables. When these transformations are applied, an input or output BIST signal is inserted in the behavior; in the structural domain, this corresponds to the insertion of a primary input or output register. However, in this scheme no *internal* BIST variables are inserted in the behavior. This means that any binding of internal

variables into registers by the synthesizer will not be affected by the BIST insertion; therefore the binding can be done independently of the BIST insertion. The inserted SELECT nodes are implemented using multiplexers that bring in a new primary input; ESCAPE nodes are implemented with the addition of new primary output. Note that these new primary inputs and outputs do not require the use of additional I/O pins; these new primary inputs and outputs are connected only to the TPGR and the MISR used for test.

The testability insertion process begins with the design behavior, and, using behavior-for-test-transformations, modifies it to produce the test behavior. The goal is to do as few modifications as are necessary to produce a behavior that is testable. We define a behavior to be *testable* if all its internal variables have randomness values above the threshold R_{tsh}, and transparency values above the threshold T_{tsh}. The values of the thresholds come from empirical analysis based on fault coverage; at the present, we are using $R_{tsh} = 0.79$ and $T_{tsh} = 0.40$.

In what follows, we first describe the overall testability insertion procedure. Next, we motivate this choice of procedure with an example illustrating the relationship between behavioral testability and testability of the synthesized structure. Finally, we end this section by describing how the overall testability insertion procedure uses behavioral-for-test transformations to fix local testability problems. A complete example of the testability insertion procedure is given with the results in Section 9.6.

9.2.2 Testability insertion procedure

The testability insertion process is iterative in nature, and uses the randomness and transparency metrics to pinpoint testability problems in the behavior. Low randomness or transparency indicates potential controllability or observability problems that may negatively impact the testability of the RTL structure synthesized from the behavior.

The overall analysis and insertion procedure is shown in Figure 9-4. It begins by computing the randomness values of all signals and variables within a behavior. Those signals and variables with randomness below the threshold R_{tsh} are considered candidates for controllable point insertion. Of the candidates, one nearest the primary inputs is selected; this is because controllable point insertion will affect the randomness not only of the insertion point, but also of all signals and variables driven by the insertion point. The exact nature of the insertion done depends on the cause of the low randomness at the selected signal or variable; the various causes of low testability are described in the last part of this section, along with the appropriate insertions. Once insertion is done to improve randomness at the selected signal or variable, the randomness values for the behavior are recomputed, and the process iterates until all randomness values are above the threshold.

The analysis defers decisions about transparency improvement until after all randomness improvement has been done. This is because a change in randomness may affect transparency, but a change in transparency has no effect on randomness. (For an explanation of this, see page 56.) Once all randomness values are above the threshold R_{tsh},

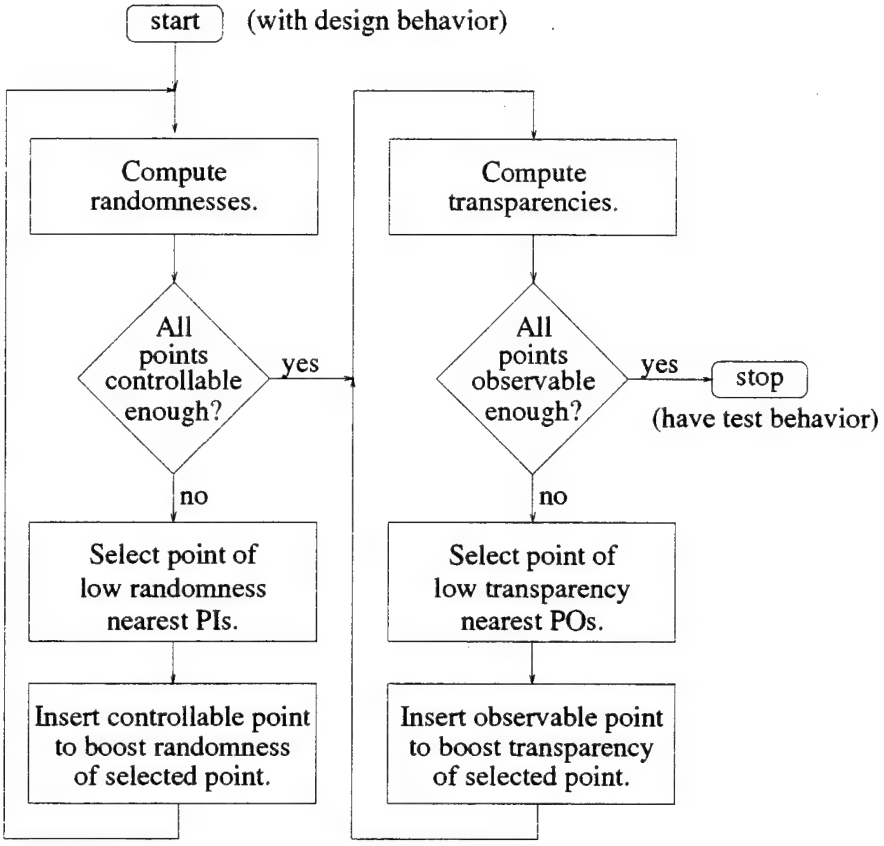


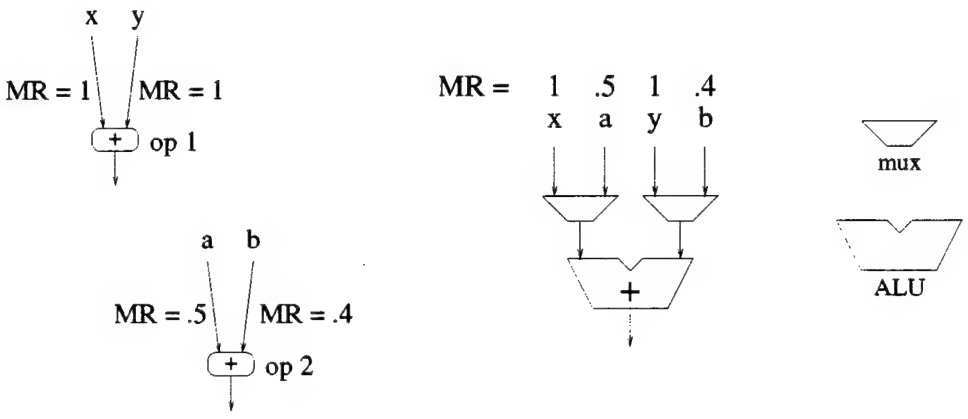
Figure 9-4. The behavioral testability insertion procedure.

transparency values are computed for all signals and variables in the behavior. All signals and variables with transparency below the threshold T_{tsh} are considered candidates for observable point insertion. Of the candidates, one closest the primary outputs is selected; this is because observable point insertion will affect the transparency not only of the insertion point, but also of all signals that drive the insertion point. As was the case with controllable point insertion, the exact nature of the insertion depends on the cause of the low transparency. Once observable point insertion has been done to

improve the transparency of the selected signal or variable, all transparencies are recomputed. The process iterates until all transparency values are above the threshold.

9.2.3 Behavioral versus structural testability

One important aspect of our work is that we require that *all* signals and variables of our final test behavior have testability metrics above the thresholds. Figure 9-5 illustrates why we make this strict requirement. Part (a) of the figure shows two addition



(a) a data flow graph fragment.

(b) the structure created when op1 and op2 are bound to the same physical adder.

Figure 9-5. Example of relationship between behavior and structure.

operations from a data flow graph. One addition (op1) has highly random data inputs, while the other (op2) does not. Part (b) of Figure 9-5 shows a physical implementation obtained by binding operations op1 and op2 to the same physical adder. From the point of view of testing the adder, there is no need to boost the randomness of the inputs to the second operation. The good quality test patterns that the adder receives when oper-

ation op1 is being performed are adequate to test the adder, so it does not matter that the test patterns received when operation op2 is being performed are of low quality. However, the focus of our approach is to test the complete datapath; we test not only the arithmetic logic units, but also the multiplexers, registers, and interconnections. Note that although the adder in Figure 9-5 will be fully tested, the input multiplexers will not; without some kind of testability insertion, one input to each multiplexer has low randomness, and so the test patterns that the multiplexers receive are of low quality. For this reason, our methodology insists that insertion be done to enhance every low testability signal or variable in a behavior. This approach has another benefit, in addition to the fact that it allows a complete test of the datapath as a whole; it also lends itself well to a pre-synthesis approach, because the testability insertion needed is independent of the choices made during binding. This means that our testability insertion approach does not require a special high level synthesizer; it is appropriate for use with any general purpose high level synthesis tool.

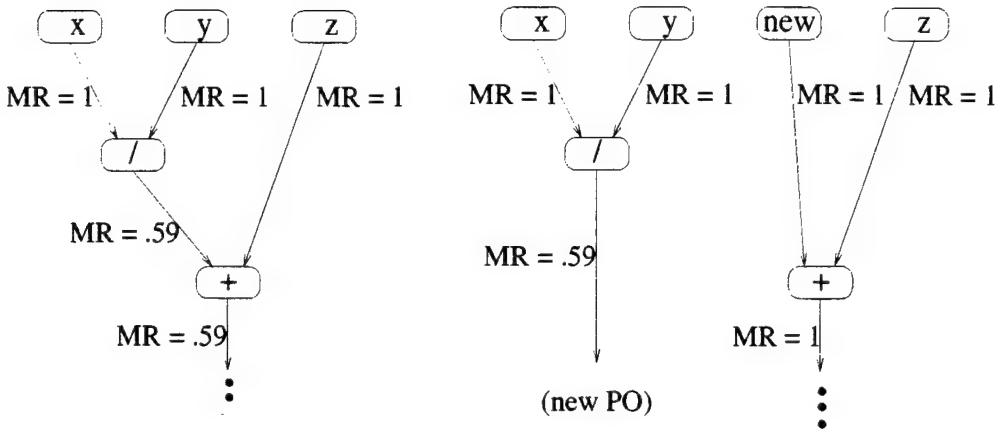
9.2.4 Three causes of low testability

This subsection describes how insertion is done to improve the testability of a signal or variable. Suppose that a signal or variable has low testability, which we define as having either randomness below the threshold R_{tsh} or transparency below the threshold T_{tsh}. In some cases, a SELECT or ESCAPE transformation is used directly at the point of unacceptable testability to improve the controllability or observability. In other cases, it is more beneficial to do insertion at a signal or variable slightly removed from

the unacceptable point. We now use examples to show the three possible causes of low testability and the proper insertion for each.

Figure 9-6(a) shows low randomness at a variable due to an inherent inability of a function to transfer randomness. Although the inputs to the integer divider are completely random, the nature of division is such that the output has poor randomness. In this case, there is no way to remove the point of low randomness entirely, but we can make sure that the low randomness signal is not used to provide test patterns for any part of the circuit. Thus, if our original design behavior contains a structure like that of Figure 9-6(a), we use a SELECT transformation to provide direct control of the low randomness variable downstream of the divider during test. In doing so, we must also use the ESCAPE transformation to insert an observable point, so that we can observe the response of the behavior upstream of the divider; otherwise, this part of the behavior will "dead-end" when the circuit is in test mode. Figure 9-6(b) shows the proper test behavior for this example. The design-and-test behavior will behave as shown in part (a) of Figure 9-6 when in design mode, and as shown in part (b) of the figure when in test mode. Note that while the low randomness variable still exists in the data flow graph, it no longer provides test patterns to any part of the behavior.

Figure 9-7(a) shows low randomness at a variable due to correlation among the inputs to the operation driving the variable. The figure shows a multiplier being used to compute the square function; the output of the multiplier has low randomness because the square function has a limited output space. In this case, it is most beneficial to use a

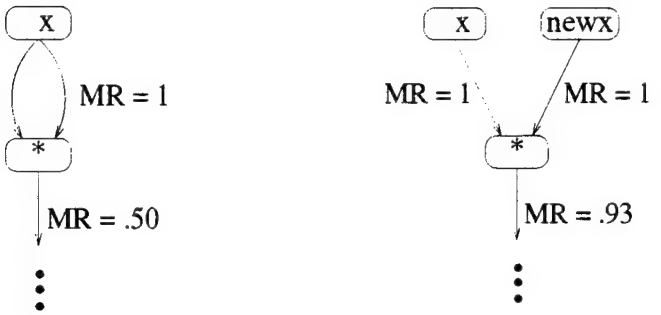


(a) original data flow.

(b) data flow after insertion of a controllable/observable pair.

Figure 9-6. Insertion when functionality causes low randomness.

SELECT transformation to break the correlation by providing a new, independent signal at one operation input during test mode. The data flow of the transformed behavior in test mode is shown in Figure 9-7(b). Note that by breaking the correlation, all low randomness variables are removed, and all operations receive high quality test patterns. There are two reasons for using insertion to break the correlation, rather than doing the insertion directly at the point of low randomness, as we did in the previous example. The first is that by removing the correlation, we can improve the testability metrics by inserting a single controllable point, rather than the controllable/observable pair needed in the previous example. The second stems from the fact that the correlation makes it difficult to test the multiplier. Although in the original behavior the two inputs to the multiplier have high randomness values, their high degree of correlation severely restricts the test patterns received by the multiplier.



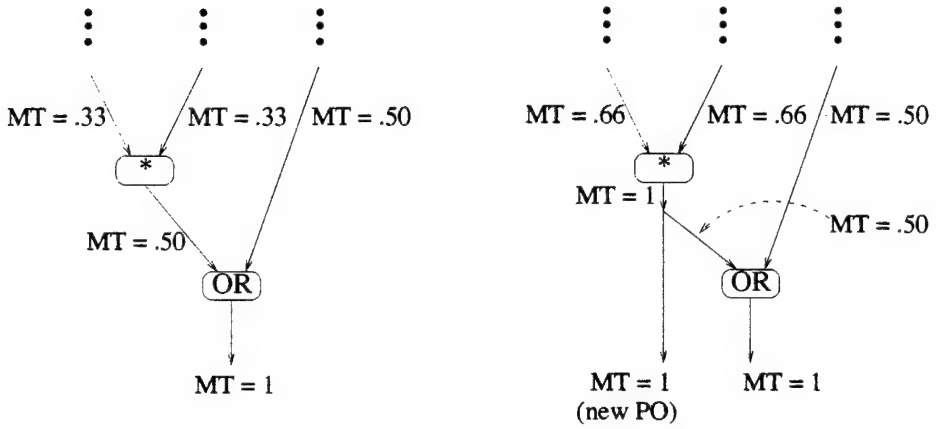
(a) original data flow.

(b) data flow after insertion of a controllable point.

Figure 9-7. Insertion when correlation causes low randomness.

Figure 9-8(a) shows low transparency at a variable that is due neither to functionality nor correlation. Here, the low transparency is caused by gradual degradation of transparency as we move up the data flow graph. The primary output of the graph is directly observable, and therefore has transparency one. One level up, the transparency drops to a value of 0.50; although this value is above the threshold, it is low enough to cause unacceptable transparency one level beyond that. In this case, it is most beneficial to use an ESCAPE transformation to do the insertion at the variable of medium transparency, as shown in Figure 9-8(b); in this way, all transparencies are brought above threshold with a single additional observation point. In contrast, if insertion is done directly at the low testability points, two observable points are needed. Also, when insertion is done directly at a low testability point, the low testability point is not completely eliminated; this is shown in Figure 9-9. The insertion of the observable point adds a fanout branch to the behavior. While the source of the fanout branch has enhanced observability, one branch still has low observability. At the register transfer

level, this can correspond to a multiplexer with poor observability, as shown in part (c) of Figure 9-9.



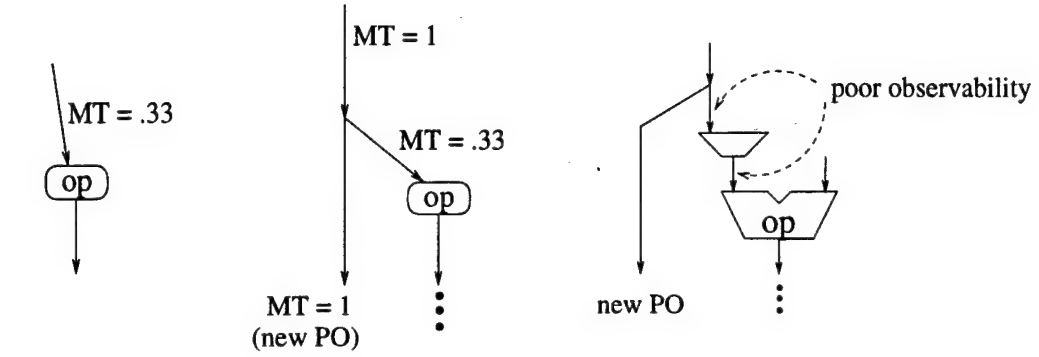
(a) original data flow.

(b) data flow after insertion of an observable point.

Figure 9-8. Insertion when gradual degradation causes low transparency.

9.3 *Structural Testability Insertion for the Controller*

While it is possible to evaluate the testability of the datapath in the behavioral domain, we do insertion for the controller and the interface between the controller and the datapath in the structural domain. The controller is implemented as a finite state machine, as shown in Figure 9-10. We improve the testability of the controller by adding circular BIST capabilities to the state flip-flops. During test, the state flip-flops linked



(a) original data flow. (b) data flow after insertion. (c) structural implementation.

Figure 9-9. Observable point insertion at the point of low transparency.

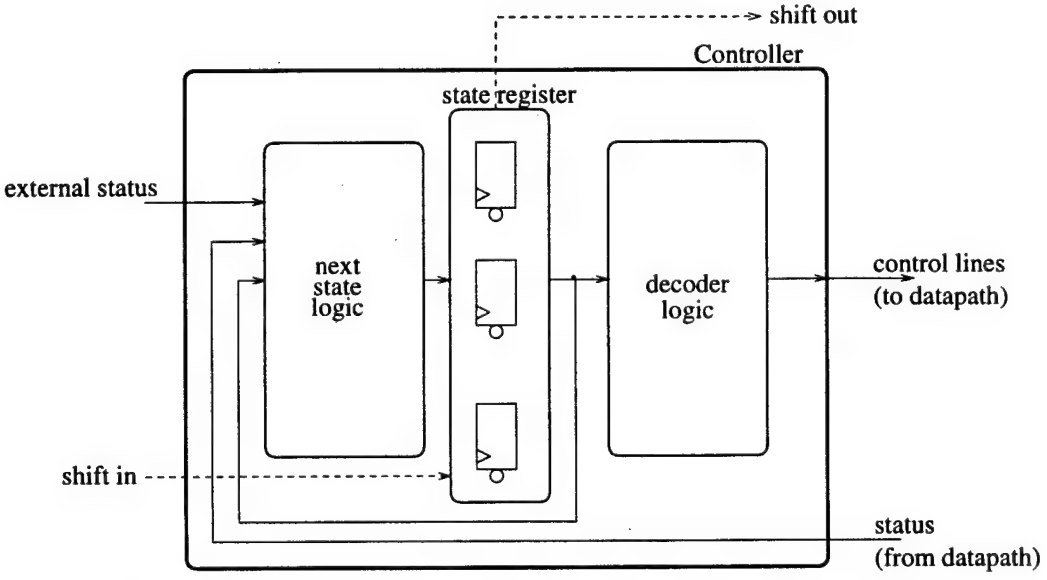


Figure 9-10. Finite state machine implementation of the controller.

together into a chain that can be configured as either a circular BIST register or an ordinary serial shift register.

In addition, we do some structural test point insertion at the interface between the controller and the datapath. The interface consists of the signals created by the controller to control the datapath, and includes both multiplexer select controls and register load controls for the datapath. The interface is tested while the datapath is tested using the behavioral test scheme described in Section 9.1. Thus, the interface lines are observed *through* the datapath while the datapath is operating in its design mode.

We decide where test point insertion is necessary in the interface by devising observability metrics for the interface control lines. A problem on a multiplexer select line is quite easy to see, as it causes the datapath to operate on incorrect data during one or more control steps. For this reason, we assign all multiplexer select lines transparency values of one.

A problem on a register load line can be more difficult to detect. For example, suppose that a register load line is stuck active, so that the register is inadvertently loaded at each control step. This will only be noticed if the inadvertent loads corrupt data before the data can be used, i.e., if the register is asked to hold data for more than one control step and is unable to do so because of the inadvertent loads. Thus, the transparency of the register load lines depends on the pattern of reads and writes for the registers. The transparency metrics are computed by combining structural information about how the

signals and variables of the behavior are bound to the registers of the datapath with behavioral information about when the signals and variables are written and read in the scheduled data flow graph. We use the scheduled data flow graph fragment of Fig-

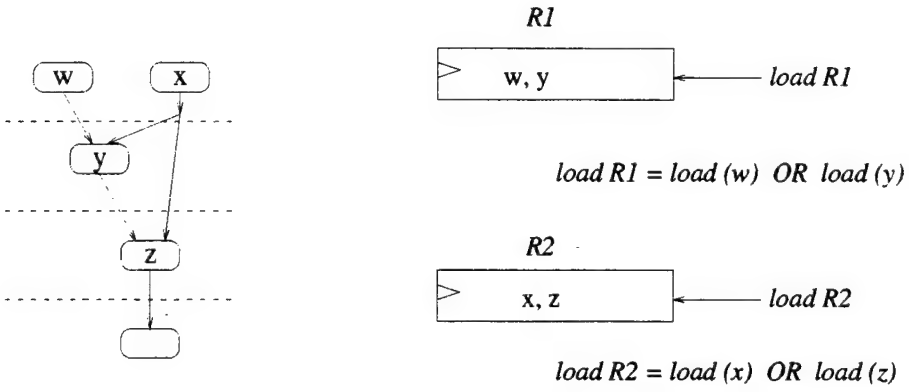


Figure 9-11. A scheduled data flow graph fragment.

ure 9-11 as an example. First, all the signals and variables of the scheduled data flow graph are divided into two classes: those that are used only in the control step after which they are created, and those which are used two or more control steps after they are created. In our example, variables *w*, *y*, and *z* are of the first class, and *x* is of the second class. If all signals and variables bound to a register are of the first class, the load line for that register is assigned a transparency value of zero. If at least one signal or variable bound to a register is of the second class, the load line for that register is assigned a transparency value of one. Suppose that in our example, *w* and *y* are bound to one register R_1, and *x* and z are bound to another register R_2. The transparency value of R_1 is then zero, while the transparency value of R_2 is one. The reason that it is

impossible to see a stuck-active fault on the load line for register R_1 is that R_1 is never asked to hold its value.

For the register load lines with transparency values of zero, structural observable point insertion is done. An additional primary output is added to the datapath to watch one bit of the register, so that inadvertent loads can be seen. For the example, we should watch one bit of register R_1. This brings the transparency of the load line for register R_1 up from a value of zero to a value of one, so that we can see whether the register's load line is working properly.

9.4 Overall Test Scheme

Our goal is to ensure testability of the *complete* physical implementation of a behavior, including both datapath and controller, and the interface between the two, as shown in Figure 9-12. We accomplish the overall test by using three test sessions. The first test session is designed to catch faults on the reset lines of the state flip-flops; these faults can prevent proper initialization of the state, causing the circuit to behave unpredictably. This short test session configures the state flip-flops in the controller in a serial shift chain. It starts by shifting in a deterministic pattern that is the complement of the reset state. Then, it resets the flip-flops, and shifts out the state. In this way, it is possible to detect if any state flip-flop does not reset properly.

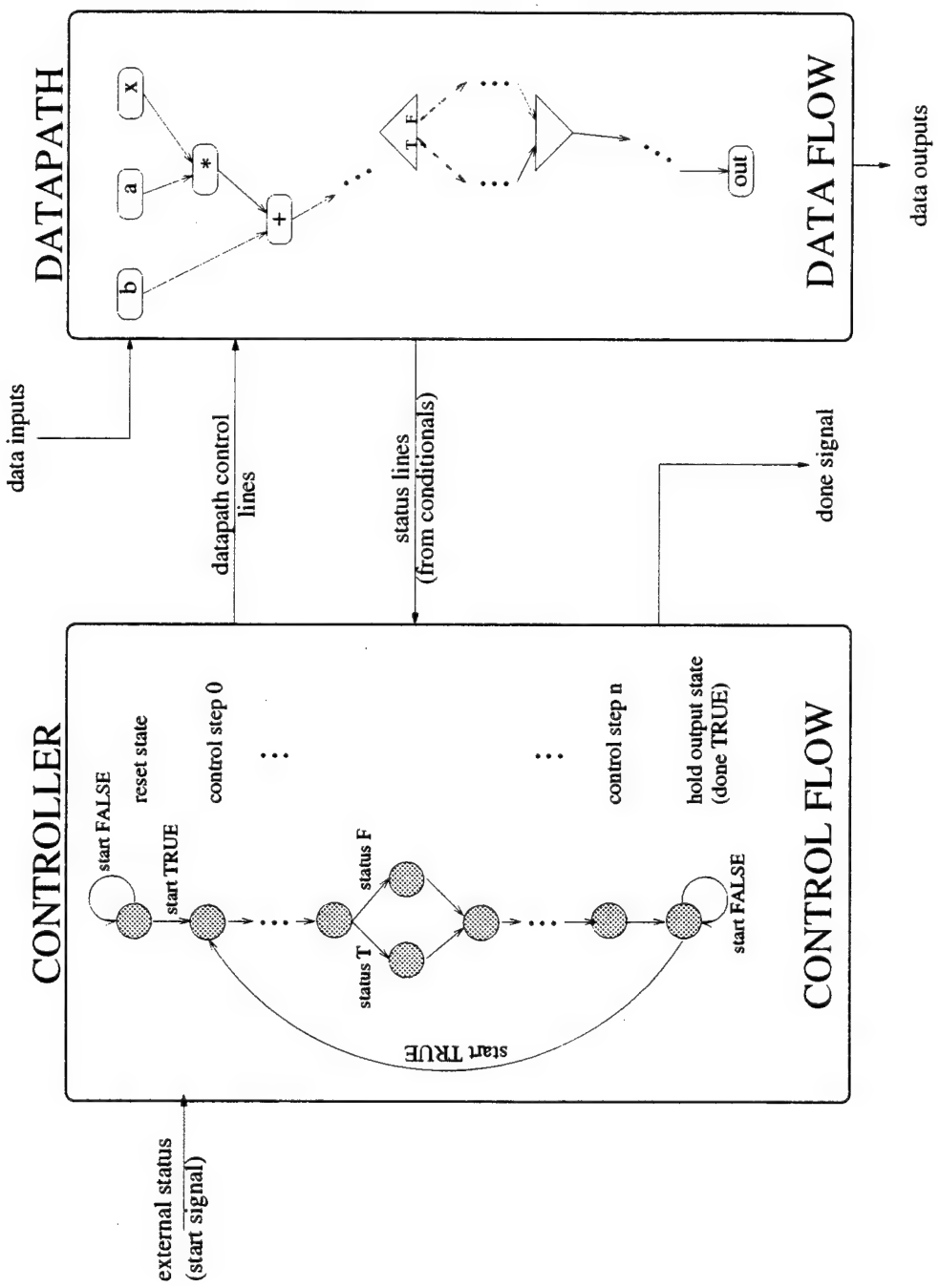


Figure 9-12. A datapath / controller pair.

The second test session uses the behavioral test scheme to detect faults within the datapath, which consists of registers, arithmetic logic units, and multiplexers. During this session, we operate the controller in normal mode, so that it creates the proper control signals to operate the datapath according to the behavior. We then use a TPGR to provide test patterns for the datapath, and an MISR to observe the outputs of the datapath. Note that it is necessary to run the datapath in both design and test mode; while only test mode is necessary to test the main part of the datapath, it is necessary to use design mode to fully test the multiplexers that implement the behavioral switch between the two modes. For this reason, we drive the behavioral mode control (test control signal) from the TPGR.

The third test session is designed to detect faults in the combinational logic of the controller. This includes both the logic that determines the next state of the controller from the current state, and the logic that decodes the proper control signals from the current state. For this test session, we configure the state register of the controller into a circular BIST register. The shift input of the register is driven by the TPGR, and the shift output of the register is observed by the MISR [Davi94]. All outputs of the controller are observed by the MISR, and all inputs of the controller, including the status and external status lines, are driven by the TPGR.

The order of the test sessions is not important, and was chosen for simplicity of implementation. We chose to test the reset faults first because doing so requires a deterministic pattern, and it was easy to design our TPGR so that the deterministic pattern we

need is shifted out first. Circuitry on chip is used to create the proper control signals to move from test session to test session.

9.5 *Background for Experiments*

The rest of this chapter is devoted to examples of our behavioral BIST insertion scheme. For each example, we start with a design behavior written in VHDL. All of the examples presented in this chapter use four bit wide signals. We apply our behavioral BIST insertion procedure to the design behavior to derive a design-and-test behavior. Next, we synthesize the design-and-test behavior, following the steps of the normal ASIC design flow (see Figure 2-10 on page 24). We also synthesize the original design behavior directly, for the purposes of comparison.To underscore the point that our BIST insertion procedure is not designed specifically for use with a particular high level synthesis system, we use two different systems, the *SYNTEST high level synthesis system* developed at Case Western Reserve University [HPCN92] and the *Behavioral Design Assistant* (BdA) developed at the University of California at Urvine [RaGa94]; thus, we are able to conclude that the success of our method does not rely on using a specific style of design.

When high level synthesis is complete, we have a register transfer level datapath and control flow in the form of a state diagram. Logic level synthesis is done using the COMPASS Design Automation suite of tools [CODA92], using a finite state machine

implementation for the controller. At this point, structural circular BIST insertion is done within the controller for the design-and-test version, as described in Section 9.3. In addition, the testability metrics for the interface control signals are computed, and additional test points are added where necessary to improve the observability of the interface.

Fault coverage curves are found for the resulting logic level circuits using AT&T's GENTEST fault simulator, and the curves for the design-and-test versions are compared to curves for the original design. The probability of aliasing within the MISRs is neglected. Although the datapath and the controller are tested together as a unit, the fault coverage results are separated into two curves; this is done to make clear how much of the fault coverage increase comes from the behavioral insertion in the datapath, and how much comes from the structural insertion in the controller.

In order to understand the cost of our method, we supply area and performance figures for both the design and design-and-test versions of the examples. Both area and performance are derived from a layout level version of the circuits, synthesized from the logic level using the COMPASS Design Automation suite of tools. Area is expressed as a transistor count. Performance is captured by the critical delay, which is the delay along the slowest combinational path in a circuit. Critical delay determines the top speed at which a circuit can be clocked. For all area and performance figures, overhead is given by:

$$\text{overhead} = \frac{v_{d\&t} - v_d}{v_d},$$

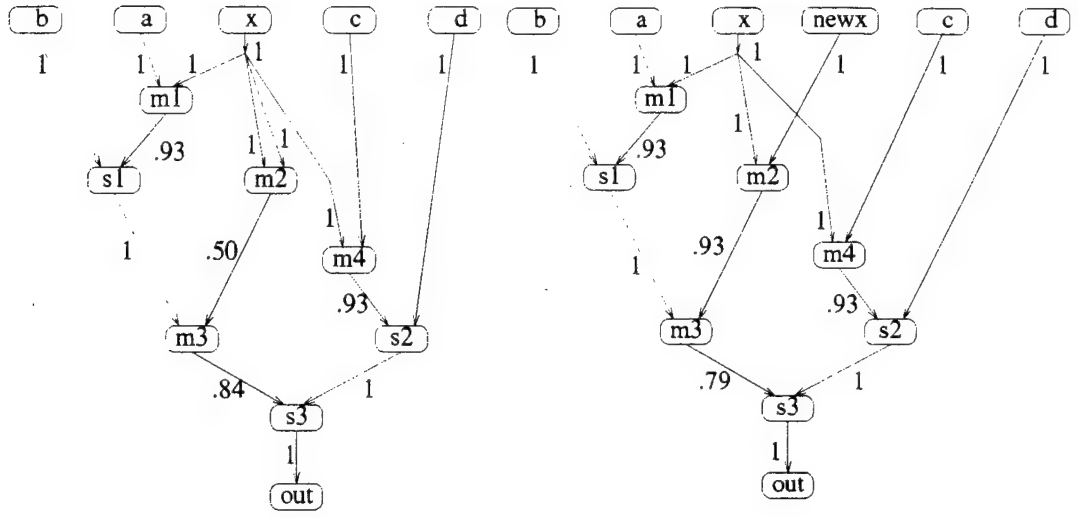
where $v_{d\&t}$ is the value for the design-and-test version of the circuit, and v_d is the corresponding value for the original design.

9.6 *Example One: A Polynomial Evaluator*

We demonstrate our iterative method for deriving a testable behavior from an original behavior on an example that evaluates the third degree polynomial $ax^3 + bx^2 + cx + d$. The first step of our method requires the computation of the randomness of the variables in the behavior; the randomness values are shown in Figure 9-13(a), superimposed on a data flow graph for the polynomial. One randomness value, at m_2, falls below the threshold of $R_{tsh} = 0.79$. Controllable point insertion is used to improve the randomness at m_2; in this case, since the low randomness is caused by correlation between the inputs of m_2, we use insertion to break the correlation, in exactly the same way described in Figure 9-7 of Section 9.2. Figure 9-13(b) shows the data flow graph after the insertion; for this graph, all randomness values are above the randomness threshold.

At this point, the method considers the transparency metrics. Figure 9-14(a) is the same data flow graph as Figure 9-13(b), but this time the transparency values for each signal and variable are shown. Two signals, at a and the leftmost branch of x, fall

design behavior *intermediate behavior*

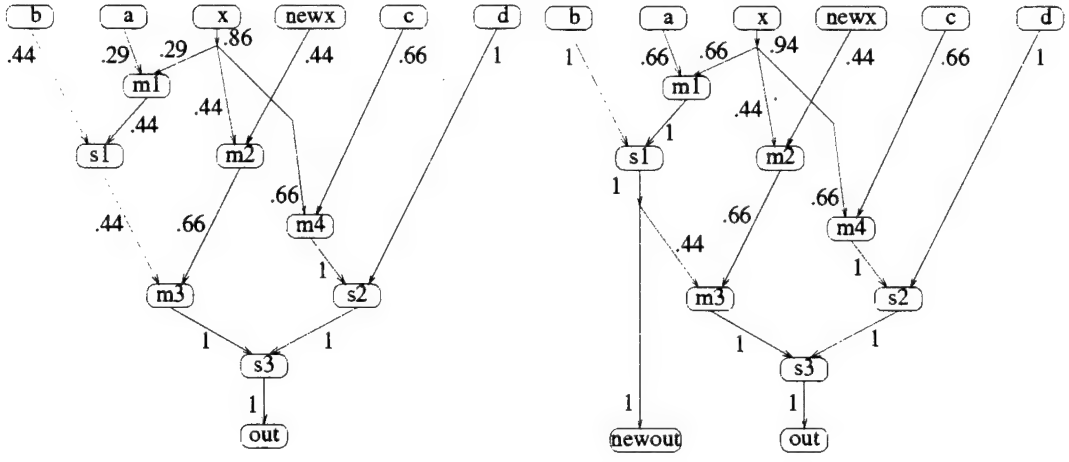


(a) before controllable point insertion.

(b) after controllable point insertion.

Figure 9-13. The data flow graph for the polynomial evaluator with annotated randomness values.

intermediate behavior *test behavior*



(a) before observable point insertion.

(b) after observable point insertion.

Figure 9-14. The data flow graph for the polynomial evaluator after controllable point insertion with annotated transparency values.

below the threshold of $T_{tsh} = 0.40$. Note that it is important to consider branches separately, as they have different transparency properties. This time, the cause of the low testability is a gradual degradation of transparency, like that described in Figure 9-8 of Section 9.2. The transparency at *s1* and *m1* is above the threshold, but low enough to adversely affect the transparency further up. Here, we improve the transparency by inserting an observable point at *s1*. The final data flow graph for the test behavior is shown in Figure 9-14(b), with all transparency values above the threshold. Note that since observable point insertion does not affect randomness values, this final data flow graph also has all randomness values above the threshold (as shown in Figure 9-13(b)).

The actual behavior synthesized is a merging of the original behavior (the *design behavior*) of Figure 9-13(a) and the derived testable behavior (the *test behavior*) of Figure 9-14(b). This merged behavior, the *design-and-test behavior,* is shown side by side with the original in Figure 9-15. The triangles in the design-and-test behavior denote divergence and merging of control. Which way control flows after the diverge node depends on the value of the test mode input; one path is for the design behavior, and the other path is for the test behavior. VHDL descriptions for the original design behavior and the design-and-test behavior are shown in Figure 9-16; the diverge and merge nodes of the data flow graph become an *if* statement in VHDL.

The design-and-test behavior of Figure 9-16 was synthesized into a datapath at the register transfer level (RTL) and a control flow graph (state diagrams) using the SYN-

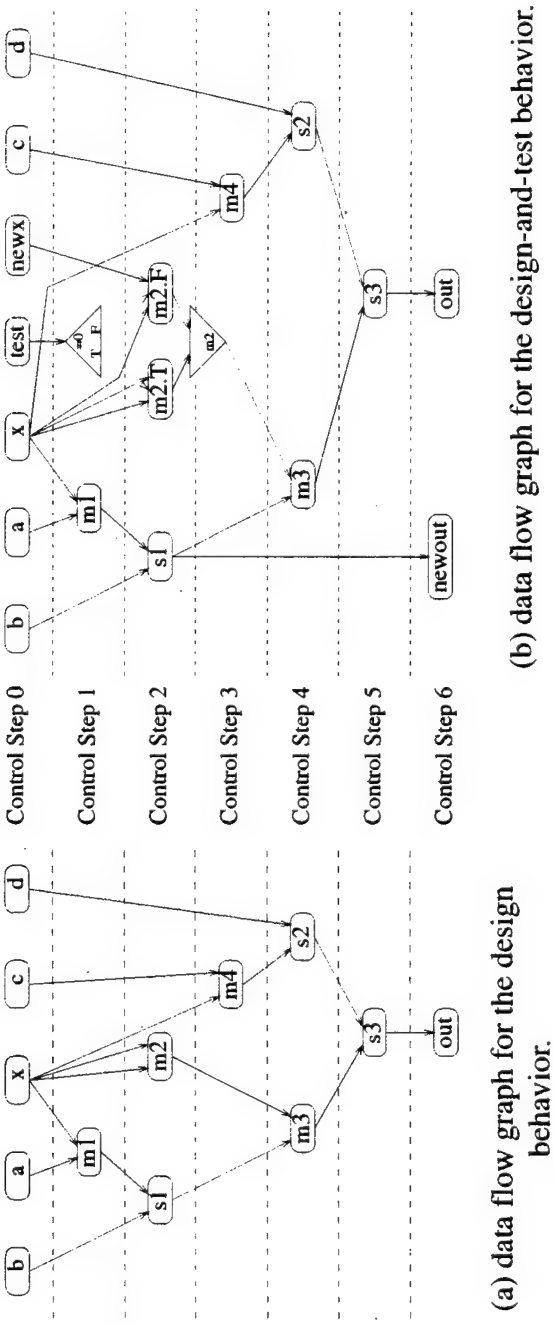


Figure 9-15. Scheduled data flow graphs for the polynomial evaluator.

```
ENTITY example IS
 PORT (a, b, c, d,
     x: IN BIT_VECTOR(3 DOWNTO 0);
     out: OUT BIT_VECTOR(3 DOWNTO 0)

     );
END example;


ARCHITECTURE behavior OF example IS
 BEGIN
  PROCESS (a, b, c, d, x)
   VARIABLE M1, M2, M3, M4,
          S1, S2, S3: INTEGER;
    BEGIN
       M1 := a * x;
       S1 := M1 + b;



       M2 := x * x;



       M3 := S1 * M2;
       M4 := c * x;
       S2 := M4 + d;
       S3 := S2 + M3;
       out <= S3;
    END PROCESS;
 END;
```

(a) design version, i.e., as originally designed.

```
ENTITY example IS
 PORT (a, b, c, d, x,
     newx: IN BIT_VECTOR(3 DOWNTO 0);
     test: IN BIT;
     out,
     newout: OUT BIT_VECTOR(3 DOWNTO 0)
     );
END example;

ARCHITECTURE behavior OF example IS
 BEGIN
  PROCESS (a, b, c, d, x, newx, test)
   VARIABLE M1, M2, M3, M4,
          S1, S2, S3: INTEGER;
    BEGIN
       M1 := a * x;
       S1 := M1 + b;
       newout <= S1;
       IF (test = '0') THEN
        M2 := x * x;
       ELSE
        M2 := x * newx;
       END IF;
       M3 := S1 * M2;
       M4 := c * x;
       S2 := M4 + d;
       S3 := S2 + M3;
       out <= S3;
    END PROCESS;
 END;
```

(b) design-and-test version, i.e., after behavioral testability insertion.

Figure 9-16. The polynomial evaluator behaviors, written in VHDL.

TEST high level synthesis system [HPCN92]. The original design behavior was also synthesized for the sake of comparison. The RTL datapath for the design-and-test behavior is almost identical to the datapath for the design behavior. The datapath for the design-and-test version is shown in Figure 9-17, with the new elements in bold. The design-and-test version has two additional registers, one each for the inserted controllable and observable points, and an additional flip-flop to hold the test mode input. In addition, one of the three-to-one multiplexers in the original datapath was changed

to a four-to-one multiplexer to pass the new controllable point, *newx*, through to the multiplier.

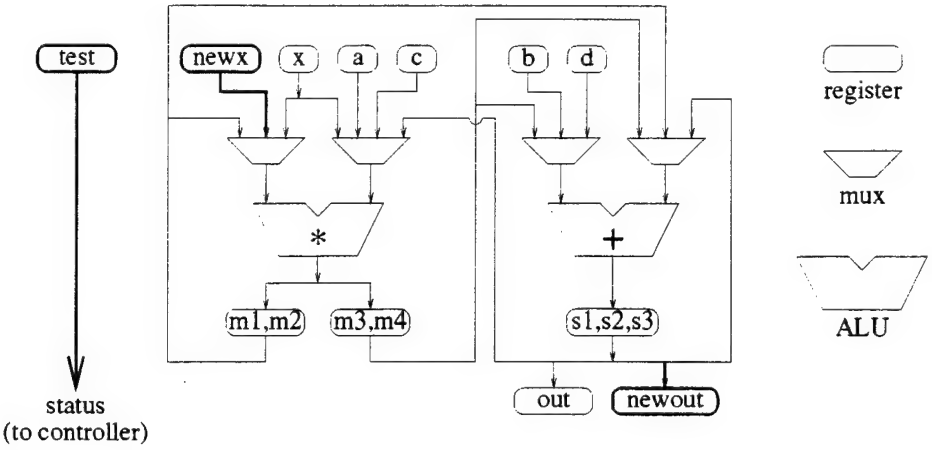
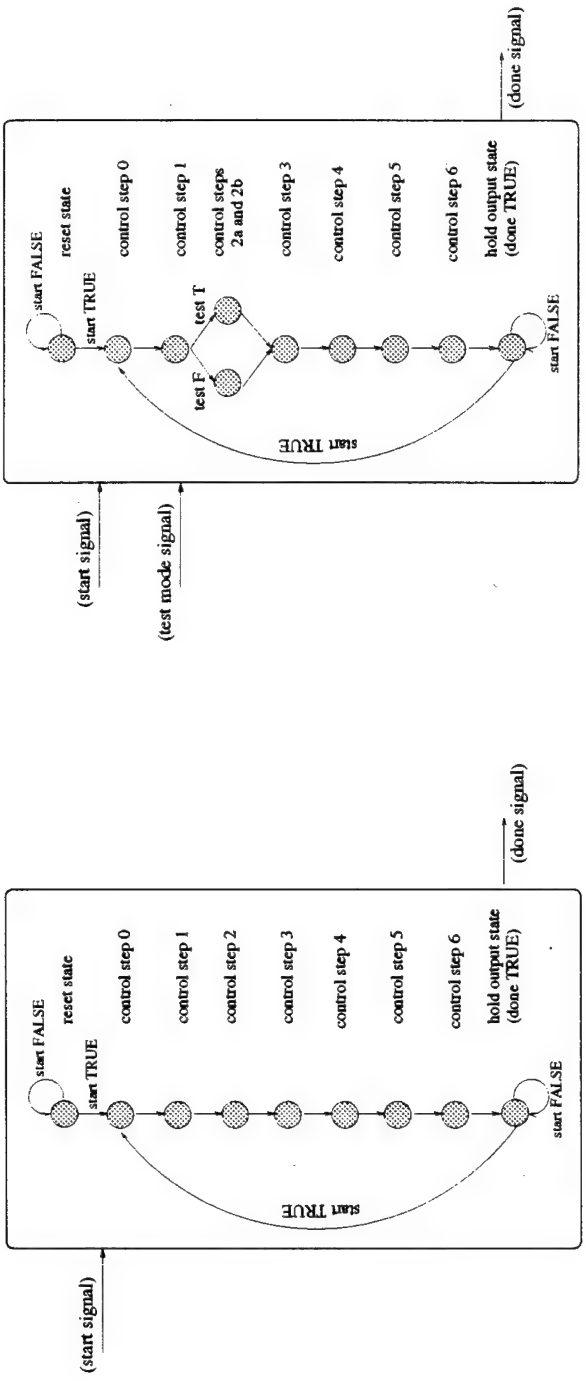


Figure 9-17. The datapath for the testable polynomial evaluator behavior as synthesized by SYNTEST, with inserted elements in bold.

The control flow graphs for the design and design-and-test controllers are also very similar, and are shown in Figure 9-18. For the original design, which has no conditionals, the control flows in a straight line from control step 0, when the primary inputs are read, to control step 6, when the final output is written. For the testable design, the only difference is a branch in control step 2 that allows the controller to implement either $m2 = x \cdot x$ for design mode operation or $m2 = x \cdot newx$ for testable mode operation.

At this point, the synthesis was completed to the logic level, and structural circular BIST insertion was done within the controller. In addition, consideration of the testability metrics for the interface control signals necessitated the addition of some

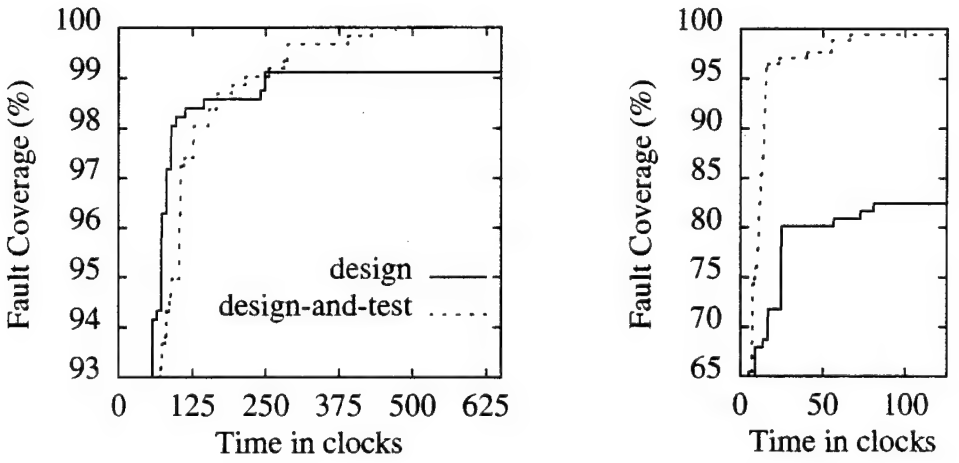


(a) control flow for the design behavior.

(b) control flow for the design-and-test behavior.

Figure 9-18. Control flow graphs for the polynomial evaluator.

observable points at the interface between the datapath and the controller. This required watching one bit each of the register that stores *a*, the register that stores *test*, and the register that stores m_3 and m_4. Fault coverage results for the datapath and controller are shown in Figure 9-19. The results show that the circuit synthesized from the design-and-test behavior is substantially more testable. Attainable fault coverage rises in the datapath as a result of the behavioral test point insertion, and in the controller as a result of the structural test point insertion. Further, fault simulation shows that the test scheme described provides very high fault coverage for the overall circuit, both datapath and controller together.



(a) for the datapath. (b) for the controller.

Figure 9-19. Fault coverage curves for the polynomial evaluator as synthesized by SYNTEST.

Area and performance figures for the SYNTEST versions of the polynomial evaluator are shown in Table 9-1. The increase in area for the datapath is due to the addition of two registers to hold *newx* and *newout*, the addition of a flip-flop to hold the test mode,

circuit	transistor count		critical delay in ns	
	datapath	controller	datapath	controller
design	1890	446	13.13	6.35
design-and-test	2227	699	13.35	7.93
overhead	18%	50%	2%	25%

Table 9-1. Area and performance figures for the polynomial evaluator as synthesized by SYNTEST.

and the replacement of a two-to-one multiplexer with a larger three-to-one multiplexer (see the bolded elements of Figure 9-17). The increase in critical delay is due to the fact that the three-to-one multiplexer has a larger delay than the original two-to-one multiplexer.

In some cases, area can be saved in the testable datapath by not bringing in a completely new controllable point; often, a controllable signal already in the datapath can be re-used. For this example, it is possible to re-use the signal *d* instead of adding a new signal *newx*; when this is done, the area overhead for the testable datapath goes down to 9%, with no degradation in attainable fault coverage. This is particularly effective for this design style, for which each primary input requires its own register.

The increase in area and delay for the controller is due to the circular BIST insertion. The overhead for this example is quite steep, with a 50% increase in area and a 25% in critical delay. Part of the area overhead problem is that the controllers for these kinds of datapaths are very small; in this case, the controller has just three state bits and about twenty gates, so the circular BIST functionality adds a substantial percentage. Note that the controller is only a small percentage of the overall circuit; for this four

bit-wide example, it is less than $\frac{1}{4}$ the size of the datapath. Further, if an eight bit- or sixteen bit-wide version of the datapath were synthesized, the controller would not change, so the controller would become even a smaller percentage of the whole. Thus, overhead in the controller is not as large a concern as overhead in the datapath.

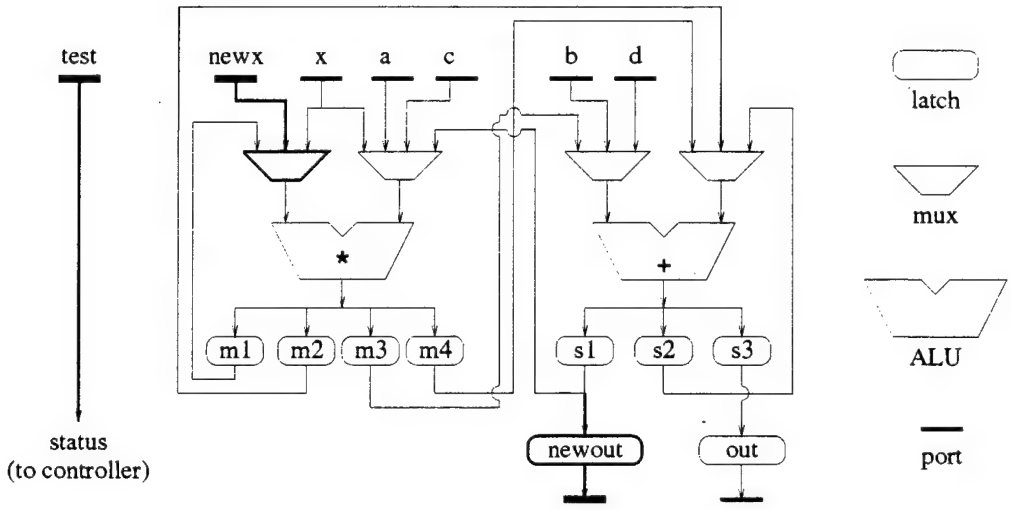
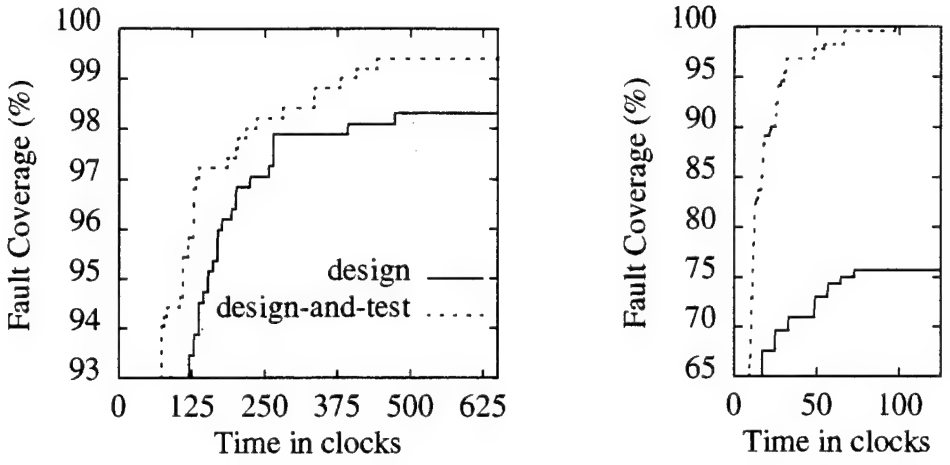


Figure 9-20. The datapath for the testable polynomial evaluator behavior as synthesized by BdA, with inserted elements in bold.

Next, the polynomial evaluator experiment was repeated, this time using the Behavioral Design Assistant (BdA) to perform high level synthesis instead of SYNTEST. The resulting datapath for the design-and-test version is shown in Figure 9-20, with the elements inserted to enhance testability highlighted in bold. The design style is significantly different from that used by SYNTEST; latches are used instead of registers, and there are no memory elements at the primary inputs of the datapath. In addition, the multiplexers used by BdA are different from those used by SYNTEST; the BdA multiplexers use one-hot encoding on their select lines. Figure 9-21 shows fault cover-

age curves; like the SYNTEST version, there is significant improvement for both the datapath and the controller. Area and performance figures for the BdA version of the polynomial evaluator are shown in Table 9-2.



(a) for the datapath. (b) for the controller.

Figure 9-21. Fault coverage curves for the polynomial evaluator as synthesized by BdA.

circuit	transistor count		critical delay in ns	
	datapath	controller	datapath	controller
design	1264	561	12.54	8.61
design-and-test	1364	921	12.66	11.18
overhead	8%	64%	1%	30%

Table 9-2. Area and performance figures for the polynomial evaluator as synthesized by BdA.

9.7 Example Two: A Differential Equation Solver

Figure 9-22 shows the data flow for a second example that implements a differential

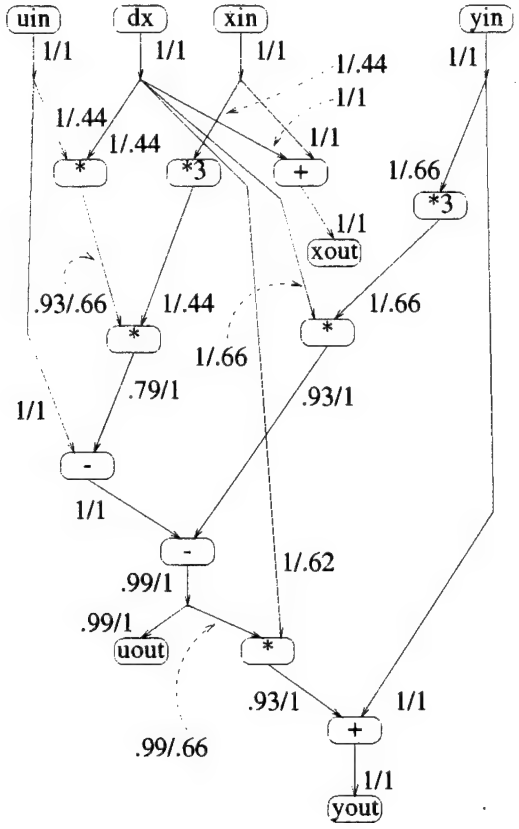
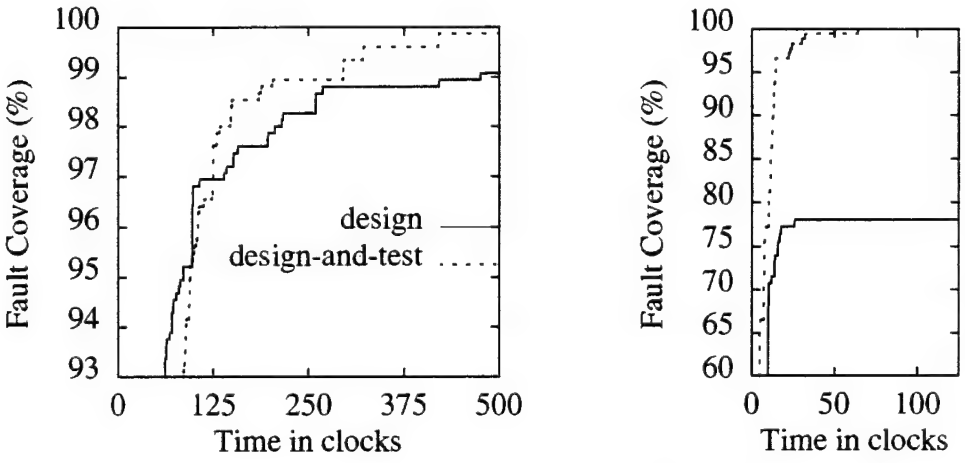


Figure 9-22. The data flow graph for the differential equation solver, with annotated randomness / transparency pairs.

equation solver, a standard high level synthesis benchmark [GDWL92].[1] The annotated randomness and transparency values show that for this behavior, no behavioral test point insertion need be done; all testability values are above the thresholds. Thus, the difference between the original and testable versions for this example is the struc-

1. The arithmetic logic units marked "*3" multiply their single input by three.

tural insertion done within the controller and at the interface between the controller and the datapath. Fault coverage curves for the design and design-and-test versions of the differential equation solver are shown in Figure 9-23. The difference in fault coverage between the design datapath and the design-and-test datapath is due to the addition of some test points at the interface between the controller and datapath to increase the observability of the interface, according to the rules given in Section 9.3 (see Figure 9-11).



(a) for the datapath. (b) for the controller.

Figure 9-23. Fault coverage curves for the differential equation solver as synthesized by SYNTEST.

Area and performance figures for the differential equation solver are given in Table 9-3; since no insertion is done within the datapath, there is zero overhead for that part of the circuit. The overheads for the controller are very similar to those in the polynomial evaluator example.

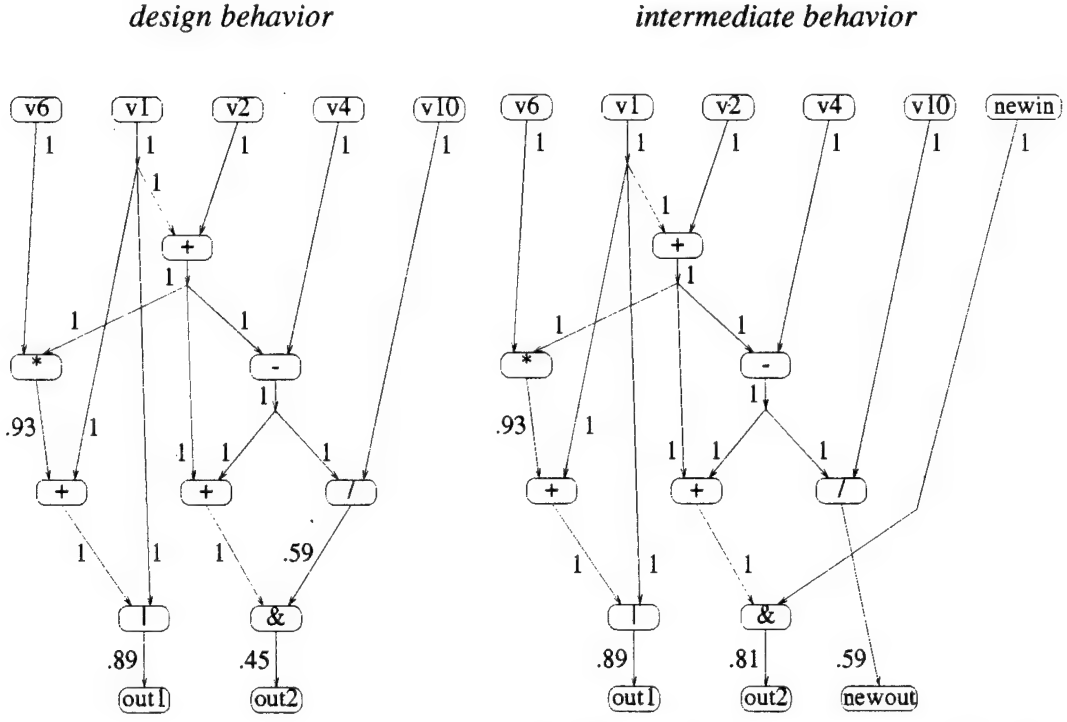
circuit	transistor count		critical delay in ns	
	datapath	controller	datapath	controller
design	2373	522	12.98	6.44
design-and-test	2373	728	12.98	7.84
overhead	0%	39%	0%	22%

Table 9-3. Area and performance figures for the differential equation solver as synthesized by SYNTEST.

9.8 *Example Three: The Facet Example*

Our final example is another high level synthesis benchmark called the facet example [GDWL92]. Its operations include addition, multiplication, division, and the logical operations AND and OR. Figure 9-24 and Figure 9-25 show the steps of the behavioral test point insertion. Part (a) of Figure 9-24 shows the data flow graph for the facet example with randomness values annotated on each edge. Two edges, the ones coming from the division and the logical AND operations, have randomness values below the threshold of 0.79. Of these two candidate edges, the one closest the primary inputs is selected for improvement. In this case, the cause of low randomness is an inherent inability of the integer divider to transfer the randomness at its inputs to its output. Following the testability insertion guideline of Figure 9-6, one controllable / observable point pair was inserted to slice the data flow graph at the output of the divider. The new data flow graph is shown in Figure 9-24(b) with new randomness values. Note that the insertion was enough to bring the randomness at the output of the logical AND operation up above the threshold.

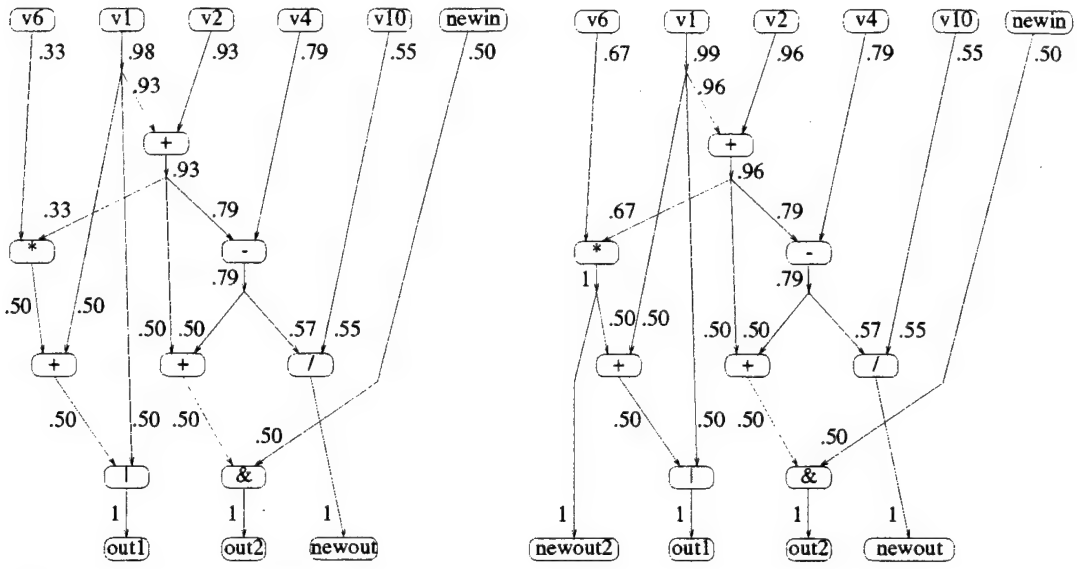
At this point, we are done with the randomness part of the behavioral test point insertion procedure, and we move on to consider transparency. Figure 9-25(a) shows the same data flow graph as Figure 9-24(b), this time annotated with transparency values. Two signals have transparency below the threshold of 0.40; here, the cause is a gradual degradation in transparency. An observable point is inserted at a node of medium transparency to fix the problem; the resulting data flow graph is shown in Figure 9-25(b), with all transparency values above the threshold. At this point, we are done with our behavioral test point insertion procedure, and the data flow graph of Figure 9-25(b) is our derived test behavior.



(a) before test point insertion.

(b) after the insertion of a controllable / observable point pair.

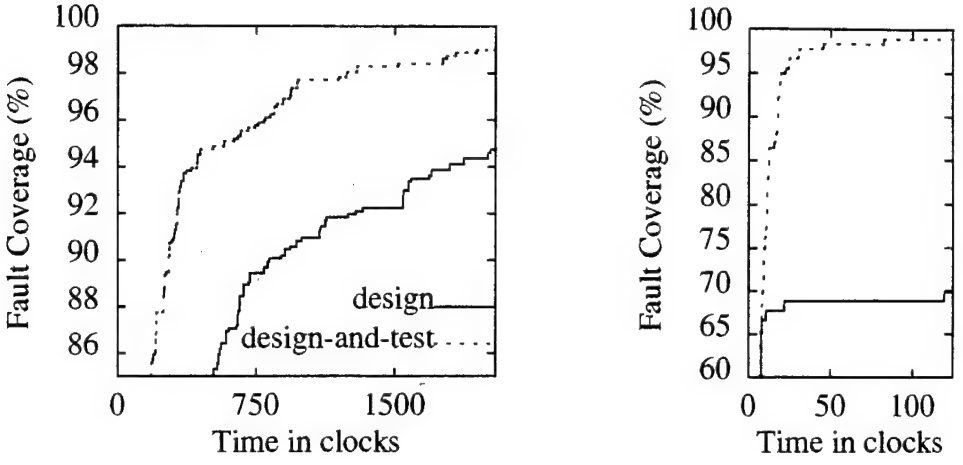
Figure 9-24. The data flow graph for the facet example with annotated randomness values.



(a) before observable point insertion. (b) after observable point insertion.

Figure 9-25. The data flow graph for the facet example with annotated transparency values.

The resulting fault coverage curves for the facet example synthesized using SYNTEST are shown in Figure 9-26. There is a substantial gain in fault coverage for the datapath due to the behavioral test point insertion; there is also a substantial gain for the controller due to the structural circular BIST insertion. Area and performance figures for the SYNTEST version are shown in Table 9-4. The facet example was also run using BdA instead of SYNTEST for the high level synthesis. Fault coverage curves for this experiment are shown in Figure 9-27, with area and performance figures in Table 9-5. The BdA version shows results similar to the SYNTEST version, despite the difference in design style. Area and performance overheads for both the SYNTEST and BdA versions are similar to those obtained for other examples.



(a) for the datapath. (b) for the controller.

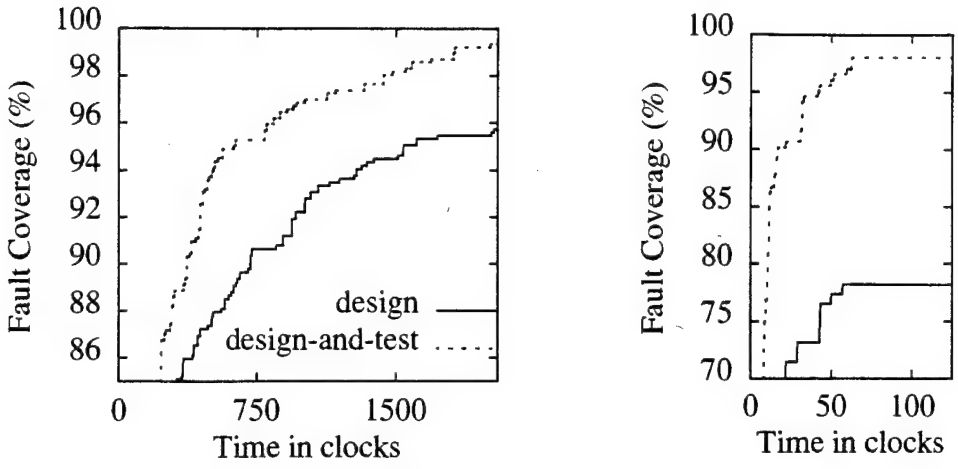
Figure 9-26. Fault coverage curves for the facet example as synthesized by SYNTEST.

circuit	transistor count		critical delay in ns	
	datapath	controller	datapath	controller
design	2927	408	11.74	5.59
design-and-test	3410	717	11.86	7.98
overhead	17%	76%	1%	43%

Table 9-4. Area and performance figures for the facet example as synthesized by SYNTEST.

9.9 *Summary*

In conclusion, we have presented a methodology for testability insertion in datapath / controller pairs. The methodology uses behavioral analysis and insertion to enhance the testability of the datapath, and structural analysis and insertion to enhance the testability of the controller and the interface between the datapath and the controller. The



(a) for the datapath. (b) for the controller.

Figure 9-27. Fault coverage curves for the facet example as synthesized by BdA.

circuit	transistor count		critical delay in ns	
	datapath	controller	datapath	controller
design	1780	474	11.07	6.77
design-and-test	1980	856	11.07	10.67
overhead	11%	81%	0%	58%

Table 9-5. Area and performance figures for the facet example as synthesized by BdA.

combination allows us to achieve high fault coverage for the overall circuit, both data-path and controller together.

Chapter 10 *Concluding Remarks*

This dissertation develops methodologies for BIST insertion across the design hierarchy. It demonstrates that despite the vast differences among circuits at different levels of design abstraction, the basic principles of testability remain the same. Regardless of the level of design abstraction, a good test relies on the ability to deliver good quality patterns to the circuit elements, and to propagate the effect of faults to the primary outputs. Thus, the testability concepts that apply at the gate level are equally applicable at the register transfer and algorithmic levels.

Furthermore, regardless of the level of design abstraction at which testability enhancement is done, a balance must be achieved, with test quality on one side, and area and system performance on the other. To be seen in full context, a gain in fault coverage must be weighed against the price of adding extra test circuitry to the chip, and the performance lost when the system must be slowed down because test circuitry has intruded on the critical paths. Throughout this dissertation, we make this trade-off clear by placing fault coverage curves side-by-side with layout areas, transistor counts, and critical delays.

The consideration of testability can begin at any point in the ASIC design flow. It is best to consider testability as early in the design flow as possible. Ideally, this means thinking about testability before any kind of synthesis begins, when a design description is at the algorithmic level in the behavioral domain. In Chapter Nine, a methodology for adding testability to behaviors prior to high level synthesis is presented. This method for behavioral test point insertion is combined with a behavioral test scheme that provides high fault coverage for the datapath portion of the circuit. Structural test point insertion and a circular BIST scheme are added for the controller portion, to provide a high quality test of the overall datapath / controller pair.

A major strength of our methodology at the algorithmic level is the fact that it does not rely on the use of a specific design style. Experiments with two different high level synthesis systems show significant gains in fault coverage for two different design styles. Since the method is generic with regards to design style, it can be used to augment an existing synthesis process with a minimum of fuss. Another strength of the behavioral test scheme presented in Chapter Nine is the fact that the control signals needed for the datapath during test are almost the same as the ones used in normal mode. This means that the test controller can be easily embedded within the design controller, with minimal changes to the control state diagram.

Although it is best to consider testability as early as possible in the design flow, it is not always possible to start in the behavioral domain. High level synthesis from behavioral descriptions is a fairly new concept, and many circuits are still designed starting

at the register transfer level. In Chapter Eight, we present a testability insertion methodology appropriate for these circuits. Examples using the circular self-test path technique show that by using testability metrics to choose where to place test points, we can significantly improve testability, either in terms of attainable fault coverage or the number of test patterns needed to achieve high fault coverage.

There are times when even a register transfer level analysis can not be done. Some circuits are inherently gate level, and must be treated as such. Chapter Seven deals with testability insertion in gate level circuits. There, the goal is to enhance the testability of the circuit while modifying the circuit as little as possible. We show two variations on the testability insertion methodology, one based on circular BIST and the other on test point insertion using multiplexers. In both cases, the insertion is used to systematically provide good quality pseudorandom test patterns to the combinational logic of the circuit, and to combat random pattern resistance within the combinational logic. The methodology is demonstrated on a submodule of an industrial design; both the circular BIST and the test point variations show significant improvement in fault coverage.

Whether dealing with gate level, register transfer level, or algorithmic level circuits, there are some structural details that must be handled carefully when the methodology of choice is circular BIST or the circular self-test path technique. Chapter Six outlines two problems that can occur, both due to bit level correlation. The first, register adjacency, is a by-product of the order of the circular BIST test flip-flops within the path. The second is a kind of correlation inherent in the shifting nature of the path, and was

identified first in this work. When present in a circuit, these two kinds of correlation can severely degrade test quality. Identification of these problems is a necessary and major step towards being able to do circular BIST and circular self-test path insertion automatically.

The cornerstone to the BIST insertion methodologies of Chapter Seven, Eight, and Nine is the set of testability metrics presented in Chapter Four. These metrics play a major role in the circular self-test path correlation analysis of Chapter Six as well. Our controllability metrics, randomness and expected state coverage, were borrowed from previous research; our testability metric, transparency, is new to this work. The Markov chain model used to calculate the metrics, detailed in Chapter Five, is an advance over the earlier models on which it was based in terms of the way that it models correlation due to reconvergent fanout and indirect feedback.

It is important to note that although we chose in this work to use randomness, expected state coverage, and transparency to quantify the testability of BIST circuits, the BIST insertion procedures presented do not rely on the use of a specific set of testability metrics. Any appropriate set of testability metrics may be used. As seen in Chapter Three, the survey of related research, there has been a great deal of interest in high level testability metrics recently, all with a different emphasis, or designed to help attain a different goal. We believe that by changing the underlying metrics, our testability insertion procedures can be easily modified for other purposes. For example, we believe that the insertion procedures need not be limited to BIST; with an appropri-

ate set of metrics, they could be used for automatic test pattern generation (ATPG)-based testability insertion as well. We plan to test this theory using the algorithmic level methodology.

We also plan to do some additional work in the area of testing controllers. In our algorithmic level work, we use circular BIST to augment the testability of our finite state machine controllers, inserting the circular BIST at the gate level. We would like to move the consideration of testability up to a higher level of design abstraction. It is difficult to see how testability insertion could be moved all the way up to the algorithmic level, since there is no concept of a controller until the datapath is created during high level synthesis. However, it may be possible to do testability insertion directly after high level synthesis, while the controller is in the form of a state diagram. Two recent projects at other universities have looked at modifying state diagrams to make controllers more easily tested using automatic test pattern generation [ChKA92] [KaCA93] [KaCA95] [PoRe93]. We would like to explore the properties of controllers that are important to a BIST scheme, and perhaps to develop a similar methodology that is appropriate to BIST, and that blends well with the behavioral BIST scheme we are currently using to test the datapath portion of the circuit. The goal of such a methodology would be to improve on the area overheads that are necessary to do a full circular BIST insertion in the controllers; in Chapter Nine, we saw that these overheads ranged from 50% to 81% for our examples.

Bibliography

[AbBF90] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, an imprint of W.H. Freeman and Company: New York, NY, 1990.

[AgKS93a] V.D. Agrawal, C.R. Kime, and K.K. Saluja, "A Tutorial on Built-In Self-Test, Part 1: Principles," *IEEE Design & Test of Computers*, Vol. 10, No. 1, pp. 73–82, March 1993.

[AgKS93b] V.D. Agrawal, C.R. Kime, and K.K. Saluja, "A Tutorial on Built-In Self-Test, Part 2: Applications," *IEEE Design & Test of Computers*, Vol. 10, No. 2, pp. 69–76, June 1993.

[Avra91] L. Avra, "Allocation and Assignment in High Level Synthesis," *Proceedings of the 1991 International Test Conference (ITC-91)*, pp. 463–472, October 1991.

[AvMc93] L.J. Avra and E.J. McCluskey, "Synthesizing for Scan Dependence in Built-In Self-Testable Designs," *Proceedings of the 1993 International Test Conference (ITC-93)*, pp. 734–743, October 1993.

[AvMc94] L.J. Avra and E.J. McCluskey, "High-Level Synthesis of Testable Designs: An Overview of University Systems," *Digest of Papers, Test Synthesis Seminar, International Test Conference (ITC-94)*, October 1994.

[BaMS87] P.H. Bardell, W.H. McAnney and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, John Wiley & Sons, Inc.: New York, NY, 1987.

[BhJh93] S. Bhatia and N.K. Jha, "Synthesis of Sequential Circuits for Easy Testability through Performance-Oriented Parallel Partial Scan," *Proceedings of the 1993 IEEE International Conference on Computer Design (ICCD-93)*, pp. 151–154, October 1993.

[BhJh94] S. Bhatia and N.K. Jha, "Behavioral Synthesis for Hierarchical Testability of Controller/Datapath Circuits with Conditional Branches," *Proceedings of the 1994 IEEE International Conference on Computer Design (ICCD-94)*, pp. 91–96, October 1994.

[BrAS90] O. Brynestad, E.J. Aas and A.E. Vallestad, "State Transition Graph Analysis as a Key to BIST Fault Coverage," *Proceedings of the 1990 International Test Conference (ITC-90)*, pp. 537–543, October 1990.

[CaPa94] J.E. Carletta and C.A. Papachristou, "Structural Constraints for Circular Self-Test Paths," *Proceedings of the 12th IEEE VLSI Test Symposium*, Cherry Hill NJ, April 1994, pp. 87-92.

[CaPa95] J.E. Carletta and C.A. Papachristou, "Testability Analysis and Insertion for RTL Circuits Based on Pseudorandom BIST," *Proceedings of the 1995 IEEE International Conference on Computer Design (ICCD-95)*, to appear, October 1995.

[ChKA92] S.T. Chakradhar, S. Kanjilal and V.D. Agrawal, "Finite State Machine Synthesis with Fault Tolerant Test Function," *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC-92)*, pp. 562–567, June 1992.

[ChSa93] C.-H. Chen and D.G. Saab, "A Novel Behavioral Testability Measure," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12, No. 12, pp. 1960–1970, December 1993.

[ChKS94] C.-H. Chen, T. Karnik, and D. G. Saab, "Structural and Behavioral Synthesis for Testability Techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 6, pp. 777–785, June 1994.

[ChWS91a] C.-H. Chen, C. Wu and D.G. Saab, "Accessibility Analysis on Data Flow Graph: An Approach to Design for Testability," *Proceedings of the 1991 IEEE International Conference on Computer Design (ICCD-91)*, pp. 463–466, October 1991.

[ChWS91b] C.-H. Chen, C. Wu and D.G. Saab, "BETA: Behavioral Testability Analysis," *Digest of Technical Papers from the 1991 IEEE/ACM International Conference on Computer Aided Design (ICCAD-91)*, pp. 202–205, November 1991.

[ChLP92] V. Chickermane, J. Lee, and J.H. Patel, "Design for Testability using Architectural Descriptions," *Proceedings of the 1992 International Test Conference (ITC-92)*, pp. 752–761, October 1992.

[ChPa91] S. Chiu and C.A. Papachristou, "A Design for Testability Scheme with Applications to Data Path Synthesis," *Proceedings of the 28th ACM/ IEEE Design Automation Conference (DAC-91)*, pp. 271–277, June 1991.

[ChGu89] C.C. Chuang and A.K. Gupta, "The Analysis of Parallel BIST by the Combined Markov Chain (CMC) Model," *Proceedings of the 1989 International Test Conference (ITC-89)*, pp. 337–343, October 1989.

[CODA92] COMPASS Design Automation, *Fundamentals I*, November 1992.

[COOS93] W.B. Culbertson, T. Osame, Y. Otsuru, J.B. Shackleford and M. Tanaka, "The HP Tsutsuji Logic Synthesis System," *Hewlett-Packard Journal*, Vol. 44, No. 4, pp. 38–51, August 1993.

[Davi94] S. Davidson, *private communication*, July 1994.

[DeVH86] W.H. Debany, P.K. Varshney, and C.R.P. Hartmann, "On Random Test Length With and Without Replacement," *Electronics Letters*, Vol. 22, No. 20, pp. 1074–1075, September 1986.

[DePo94] S. Dey and M. Potkonjak, "Transforming Behavioral Specifications to Facilitate Synthesis of Testable Designs," *Proceedings of the 1994 International Test Conference (ITC-94)*, pp. 184–193, October 1994.

[FeSV94] V. Fernandez, P. Sanchez and E. Villar, "High-Level Synthesis Guided by Testability Measures," *presentation at the First International Test Synthesis Workshop*, May 1994.

[Gage93] R. Gage, "Structured CBIST in ASICs," *Proceedings of the 1993 International Test Conference (ITC-93)*, pp. 332–338, October 1993.

[GaKu83] D. Gajski and R. Kuhn, "Guest Editors' Introduction: New VLSI Tools," *IEEE Computer*, Vol. 6, No. 12, pp. 11–14, December 1983.

[GDWL92] D. Gajski, N. Dutt, A. Wu and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers: Boston, MA, 1992.

[GCED94] M.H. Gentil, D. Crestani, A. El Rhalibi and C. Durante, "A New High Level Testability Measure: Description and Evaluation," *Proceedings of the 12th IEEE VLSI Test Symposium*, pp. 421–426, April 1994.

[Gold79] L.H. Goldstein, "Controllability / Observability Analysis of Digital Circuits," *IEEE Transactions on Circuits and Systems*, pp. 685 –693, September 1979.

[GuKP94] X. Gu, K. Kuchcinski and Z. Peng, "Testability Analysis and Improvement from VHDL Behavioral Specifications," *Proceedings of the Third European Design Automation Conference (EURO-DAC 94)*, pp. 644–649, September 1994.

[HaPa93] H. Harmanani and C.A. Papachristou, "An Improved Method for RTL Synthesis with Testability Tradeoffs," *Digest of Technical Papers from the 1993 IEEE/ACM International Conference on Computer Aided Design (ICCAD-93)*, pp. 30–35, November 1993.

[HPCN92] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test," *Proceedings of the First European Design Automation Conference (EURODAC-92)*, pp. 402–407, September 1992.

[HaPN93] H. Harmanani, C. Papachristou and M. Nourani, "The SYNTEST Design for Test System User Manual, Release 1.0," *Case Western Reserve University Technical Report CES-93-15*, June 1993.

[HaOr94a] I. G. Harris and A. Orailoglu, "Microarchitectural Synthesis of VLSI Designs with High Test Concurrency," *Proceedings of the 31st ACM/ IEEE Design Automation Conference (DAC-94)*, pp. 206–211, June 1994.

[HaOr94b] I. G. Harris and A. Orailoglu, "SYNCBIST: SYNthesis for Concurrent Built-In Self-Testability," *Proceedings of the IEEE International Conference on Computer Design (ICCD-94)*, pp. 101–104, October 1994.

[HuPe87] C.L. Hudson and G.D. Peterson, "Parallel Self-Test with Pseudo-Random Patterns," *Proceedings of the 1987 International Test Conference (ITC-87)*, pp. 954–963, October 1987.

[JaKa93] M. Jamoussi and B. Kaminska, "Controllability and Observability Measures for Functional-Level Testability Evaluation," *Proceedings of the 11th IEEE VLSI Test Symposium*, pp. 154–157, April 1993.

[John89] B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley Publishing Company, Inc.: Reading, MA, 1989.

[KaCA93] S. Kanjilal, S.T. Chakradhar and V.D. Agrawal, "A Synthesis Approach to Design for Testability," *Proceedings of the 1993 International Test Conference (ITC-87)*, pp. 754–763, October 1993.

[KaCA95] S. Kanjilal, S.T. Chakradhar and V.D. Agrawal, "Test Function Embedding Algorithms with Application to Interconnected Finite State Machines," to appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[KiHT88] K. Kim, D. Ha and J. Tront, "On Using Signature Registers as Pseudorandom Pattern Generators in Built-in Self-Testing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 7, No. 8, pp. 919–928, August 1988.

[KoMZ79] B. Konemann, J. Mucha, and G. Zwiehoff, "Built-In Logic Block Observation Technique," *Digest of Papers from the 1979 IEEE Test Conference*, pp. 37–41, October 1979.

[KrPi87] A. Krasniewski and S. Pilarski, "Circular Self-Test Path: A Low-Cost BIST Technique," *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC-87)*, pp. 407–415, July 1987.

[KrPi89] A. Krasniewski and S. Pilarski, "Circular Self-Test Path: A Low-Cost BIST Technique for VLSI Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 8, No. 1, pp. 46–55, January 1989.

[KuPe89] K. Kuchinski and Z. Peng, "Testability Analysis in a VLSI High-Level Synthesis System," *Microprocessing and Microprogramming, the EUROMICRO Journal*, Vol. 28, pp. 295–300, 1989.

[LWJA92] T.-C. Lee, W.H. Wolf, N.K. Jha, and J.M.Acken, "Behavioral Synthesis for Easy Testability in Data Path Allocation," *Proceedings of the International Conference on Computer Design (ICCD-92)*, pp. 29–31, October 1992.

[LeJW93a] T.-C. Lee, N. K. Jha, W. H. Wolf, "Behavioral Synthesis for Highly Testable Data Paths under the Non-Scan and Partial Scan Environments," *Proceedings of the 30th ACM/IEEE Design Automation Conference (DAC-93)*, pp. 292–297, June 1993.

[LeJW93b] T.-C. Lee, N. K. Jha, W. H. Wolf, "A Conditional Resource Sharing Method for Behavioral Synthesis of Highly Testable Data Paths," *Proceedings of the 1993 International Test Conference (ITC-93)*, pp. 744–753, October 1993.

[LePa92] J. Lee and J.H. Patel, "An Instruction Sequence Assembling Methodology for Testing Microprocessors," *Proceedings of the 1992 International Test Conference (ITC-92)*, pp. 49–58, October 1992.

[LiZB93] C.-J. Lin, Y. Zorian and S. Bhawmik, "PSBIST: A Partial-Scan Based Built-In Self-Test Scheme," *Proceedings of the 1993 International Test Conference (ITC-93)*, pp. 507–516, October 1993.

[LiNB93] S.-P. Lin, C.A. Njinda and M.A. Breuer, "Generating a Family of Testable Designs Using the BILBO Methodology," *Journal of Electronic Testing: Theory and Applications*, Vol. 4, pp. 71–89, 1993.

[MaSa92] A. Majumdar and S. Sastry, "On the Distribution of Fault Coverage and Test Length in Random Testing of Combinational Circuits," *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC-92)*, pp. 341–346, June 1992.

[MuJS94a] A. Mujumdar, R. Jain and K. Saluja, "Incorporating Testability Considerations in High-Level Synthesis," *Journal of Electronic Testing: Theory and Applications*, Vol. 5, No. 1, pp. 43–55, February 1994.

[MuJS94b] A. Mujumdar, R. Jain and K. Saluja, "Behavioral Synthesis of Testable Designs," *Proceedings of the 24th IEEE International Symposium on Fault-Tolerant Computing (FTCS-94)*, pp. 436–445, June 1994.

[OrHa93] A. Orailoglu and I.G. Harris, "Test Path Generation and Test Scheduling for Self-Testable Designs," *Proceedings of the 1993 IEEE International Conference on Computer Design (ICCD-93)*, pp. 528–531, October 1993.

[PaCa95] C. Papachristou and J. Carletta, "Test Synthesis in the Behavioral Domain," *Proceedings of the 1995 International Test Conference (ITC-95)*, to appear, October 1995.

[PaCH91] C. Papachristou, S. Chiu and H. Harmanani, "A Data Path Synthesis Method for Self-Testable Designs," *Proceedings of the 28th ACM/IEEE Design Automation Conference (DAC-91)*, pp. 378–384, June 1991.

[Papo84] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, Inc.: New York, NY, 1984.

[PaBN94] I. Parulkar, M.A. Breuer and C.A. Njinda, "Extraction of a High-level Structural Representation from Circuit Descriptions with Applications to DFT / BIST," *Proceedings of the 31st ACM/IEEE Design Automation Conference (DAC-94)*, pp. 345–350, June 1994.

[PaKn89] P. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 8, No. 6, pp. 661–679, June 1989.

[PaMc75] K. Parker and E.J. McCluskey, "Probabilistic Treatment of General Combinational Networks," *IEEE Transactions on Computers*, Vol. C-24, No. 6, pp. 668–670, June 1975.

[Peng95] Z. Peng, "High-Level Test Synthesis Using Design Transformations," presentation at the *Second International Test Synthesis Workshop*, May 1995.

[PeKu94] Z. Peng and K. Kuchcinski, "Automated Transformation of Algorithms into Register-Transfer Level Implementations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 2., pp. 150–165, February 1994.

[PiKK92] S. Pilarski, A. Krasniewski and T. Kameda, "Estimating Testing Effectiveness of the Circular Self-Test Path Technique," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 11, No. 10, pp. 1301–1316, October 1992.

[PoRe93] I. Pomeranz and S.M. Reddy, "Design and Synthesis for Testability of Synchronous Sequential Circuits Based on Strong-Connectivity," *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing (FTCS-93)*, pp. 492–501, June 1993.

[POLB88] M.M. Pradhan, E.J. O'Brien, S.L. Lam and J. Beausang, "Circular BIST with Partial Scan," *Proceedings of the 1988 International Test Conference (ITC-88)*, pp. 719–729, October 1988.

[RaGa94] L. Ramachandran and D. Gajski, "Behavioral Design Assistant (BdA) User's Manual, Version 1.0," *University of California/Urvine Technical Report 94-36,* September 1994.

[SaMa91a] S. Sastry and A. Majumdar, "Test Efficiency Analysis of Random Self-Test of Sequential Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 10, No. 3, pp. 390–398, March 1991.

[SaMa91b] S. Sastry and A. Majumdar, "A Branching Process Model for Observability Analysis of Combinational Circuits," *Proceedings of the 28th ACM/IEEE Design Automation Conference (DAC-91)*, pp. 452–457, June 1991.

[SCDM93] J. Steensma, F. Catthoor, and H. De Man, "Partial Scan at the Register-Transfer Level," *Proceedings of the 1993 International Test Conference (ITC-93),* pp. 448–497, October 1993.

[Stra88] G. Strang, *Linear Algebra and its Applications*, Harcourt Brace Jovanovich, Inc.: San Diego, CA, 1988.

[StWu94] A.P. Stroele and H.-J. Wunderlich, "Configuring Flip-Flops to BIST Registers," *Proceedings of the 1994 International Test Conference (ITC-94)*, pp. 939–948, October 1994.

[Stro88] C.E. Stroud, "Automated BIST for Sequential Logic Synthesis," *IEEE Design & Test of Computers*, pp. 22–32, December 1988.

[ThAb89] K. Thearling and J. Abraham, "An Easily Computed Functional Level Testability Measure," *Proceedings of the 1989 International Test Conference (ITC-89)*, pp. 381–390, October 1989.

[ThVA94] T. Thomas, P. Vishakantaiah and J.A. Abraham, "Impact of Behavioral Modifications for Testability," *Proceedings of the 12th IEEE VLSI Test Symposium*, pp. 427–431, April 1994.

[VaOr95] M. Vahidi and A. Orailoglu, "Testability Metrics for Synthesis of Self-Testable Designs and Effective Test Plans," *Proceedings of the 13th IEEE VLSI Test Symposium*, pp. 170–175, April 1995.

[ViAA92] P. Vishakantaiah, J. A. Abraham, and M. Abadir, "Automatic Test Knowledge Extraction from VHDL (ATKET)," *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC-92)*, pp. 273–278, June 1992.

[VTAA93] P. Vishakantaiah, T. Thomas, J. A. Abraham, and M. Abadir, "AMBIANT: Automatic Generation of Behavioral Modifications for Testability," *Proceedings of the 1993 IEEE International Conference on Computer Design (ICCD-93)*, pp. 63–66, October 1993.

[WaMc86] L.T. Wang and E.J. McCluskey, "Concurrent Built-In Logic Block Observer (CBILBO)," *Proceedings of the International Symposium on Circuits and Systems*, Vol. 3, pp. 56–59, November 1986.